Introdução à Computação

Da lógica aos jogos com Ruby

GUILHERME SILVEIRA

Sumário

Versão: 18.4.26

Capítulo 1

Introdução

1.1 Programação

O profissional que cria o código por trás de um jogo, um site na internet ou um aplicativo mobile é o programador. Ele é o dono do seu, do meu, de todos os computadores. Não queremos ser apenas usuários, que ligam o computador e são obrigados a seguir as regras estabelecidas por outros. A vontade de criar algo novo, de alterar algo existente, é o que nos torna programadores.

Toda introdução a programação envolve entender por cima como funcionam o computador e um programa. As implicações nos limites da computação são sentidos no dia a dia do programador.

A ideia por trás deste material é de cobrir o conteúdo de uma matéria de Introdução à Programação do primeiro ou segundo semestre de uma faculdade de Análise de Sistemas, Ciência da Computação, Engenharia etc. Portanto não pretendemos abordar tópicos muito específicos de uma linguagem

ou outra, mas sim tratar de questões sobre como um programa é criado, qual a sua base, como detectar problemas e transformar nossas ideias e soluções em código.

Vamos nos focar em conceitos extremamente importantes como construção da lógica, pilha de execução, simulação de código e a recursão.

Por isso não seguiremos ao pé da letra todos os idiomatismos da linguagem Ruby, mas mostraremos diversas variações de como se programar uma mesma estrutura lógica, tentando indicar vantagens e desvantagens de cada abordagem.

Durante essa jornada criaremos três jogos distintos, um jogo de adivinhação numérica, um jogo da forca e um baseado no clássico Pacman nosso Foge-Foge.

O jogo de adivinhação permite ao jogador escolher o nível de dificuldade e ele tem que acertar o número secreto escolhido em um intervalo que depende da dificuldade. São dadas dicas em relação ao número a cada novo chute.

Já no jogo da forca, o jogador deve adivinhar uma palavra secreta lida aleatoriamente de um arquivo que funciona como um dicionário. Passaremos a armazenar quem é o grande ganhador do jogo em um arquivo local.

Por fim, no Foge-Foge veremos como implementar um jogo baseado em turnos (*turn based*) onde o herói pode andar por um mapa, lido entre diversos arquivos, com inimigos (fantasmas) e bombas que podemos usar.

No fim, você estará apto para tomar o próximo passo em sua carreira como profissional desenvolvedor, seja praticando mais, aprendendo Orientação a Objetos, estrutura de dados, algoritmos, tudo de acordo com o que deseja para si, uma carreira no mundo dos jogos, internet, mobile etc.

Todo o código pode ser encontrado no link a seguir, ele está separado de acordo com o final de cada capítulo.

http://bit.ly/1LGOznk

1.2 Para o professor

Ao utilizar esse material em sua sala de aula, oferecemos diversos exercícios que são extensões ligadas diretamente aos jogos implementados. Oferecemos

também desafios numéricos que permitem ao aluno criar um novo programa do zero.

Sinta-se à vontade para enviar sugestões de novos exercícios, sejam eles numéricos, lógicos, jogos, ou extensões ao jogo criado.

1.3 Quem sou eu

Agora que a parte mais importante nosso aprendizado já foi discutido, chegamos à minha apresentação.

Meu nome é Guilherme Silveira, trabalho com desenvolvimento de software desde meu primeiro estágio em 1996. Aprendi a programar na época aos 9 anos, meu irmão era meu professor particular, que me instigou a criar pequenos jogos em Basic. Jogos são motivos interessantes pois trazem um prazer tanto no desenvolvimento quanto a possibilidade de compartilhá-los com qualquer um de nossos familiares.

Apesar de ter trabalhado por um bom tempo com Visual Basic, C e C++, cresci como profissional utilizando a linguagem Java, que até hoje é a minha principal ferramenta, mas nunca me mantive restrito a ela apesar do gosto pessoal. Aprendi Ruby a primeira vez em 2009 e vejo a linguagem como um contraponto interessante à abordagem *type safe* e compilada do Java, o que permite entender melhor os pontos fortes e fracos de cada linguagem, e de outras que vim a aprender nos anos que se passaram.

Por gostar de ensinar computação desde criança, acabei por fundar junto ao Paulo Silveira a Caelum em 2004. De lá para cá, junto ao Maurício Aniche me tornei coordenador do Alura, onde tenho a oportunidade de criar os mais diversos cursos, efetuando extensiva pesquisa com experts em cada área. É um prazer estar sempre aprendendo com pessoas tão interessantes e cheias de conhecimento, juntá-lo e repassá-lo com a didática da Caelum nos três meios que acho mais interessantes hoje em dia: como livro, como curso online e presencial.

1.4 Agradecimentos

Dedico este livro ao meu pai, Carlos Felício, pela sua alegria, nossa felicidade, por ter acreditado em meu potencial como educador e matemático, por ter

incentivado minha independência e crescimento profissional. São centenas os pontos de minha vida pessoal e profissional onde o caminho foi caminhado por mim, mas não sem antes consultá-lo.

Um agradecimento especial ao Paulo Silveira, que mais de 20 anos atrás me ensinou o conteúdo aqui presente, e que me incentivou a escrever este livro.

Agradeço à Joviane Jardim, que tanto atualiza o material didático da Caelum sobre Ruby, respectivamente de quem e onde pude aprender *comos* e *porquês* da linguagem. Agradeço ao Fábio Akita, cujo blog serve como inspiração para entender melhor a evolução da linguagem e de seu ecossistema durante todos esses anos. Ao Fábio Kung, que muitos anos atrás me introduziu ao mundo Ruby, e me levou a perguntar pela primeira vez como a linguagem influenciava diversas características de meu código, abrindo a porta para minhas primeiras críticas de todas as linguagens.

Obrigado ao Maurício Aniche, pelas inúmeras discussões semanais que temos toda vez que começamos um novo livro, onde mais aprendo do que ensino. E obrigado ao Adriano Almeida que mais uma vez aturou meus spams sobre o conteúdo, título e viabilidade deste projeto.

Obrigado a você, aluno e professor, estudante ao se aventurar comigo pelo aprendizado de programação, professor ao compartilhar com seus amigos esse novo mundo que se abre para uma nova geração de jovens brasileiros, o mundo onde somos os donos do computador.

Capítulo 2

Prefácio por Fábio Akita

As coisas mudam muito rápido no mundo da tecnologia. Eu comecei a digitar minhas primeiras linhas de código num MSX Hotbit no meio dos anos 80, na antiga linguagem Basic.

Escrever código é muito simples, literalmente qualquer um consegue pesquisar trechos de código no Google e colar tudo junto, em um emaranhado de instruções que, com sorte, consegue processar alguma coisa útil.

Todos que pensam em tecnologia imaginam um mundo onde tudo muda constantemente, onde o que aprendemos ontem vai ser jogado fora amanhã e teremos que aprender tudo de novo. Qual o sentido em estudar demais quando sabemos que podemos simplesmente pegar alguns pedaços de código e fazer tudo aparentemente funcionar?

Recentemente observei uma chef, que realmente estudou gastronomia, cozinhando. As pequenas coisas me chamaram a atenção. Para engrossar um

caldo, até mesmos nós, programadores e nerds que mal sabem fritar um ovo, sabemos que basta colocar algumas colheres de maizena e *voilà*. Ela primeiro pegou um pouco do caldo em uma tigela e misturou a maizena bem, depois jogou na panela. Por quê? Porque se colocar a maizena diretamente ela vai empelotar.

Em um pequeno erro, ela esqueceu de colocar sal no arroz. E agora? Se fosse eu, tentaria consertar jogando sal diretamente no arroz e tentando misturar. Ia ficar uma droga, partes com sal demais, partes ainda sem sal. Ela pegou uma tigela de água, misturou o sal e daí jorrou a água salgada nesse arroz. Ficou como se não tivesse faltado sal.

O que é isso? São técnicas fundamentais, coisas que para qualquer chef é o óbvio, o nosso "hello world". Mas para mim, um completo amador, são novidades que serão muito úteis no futuro.

Programar é muito fácil, como fazer arroz. Eu sempre digo que qualquer um pode colocar ingredientes numa panela e ligar o fogo, mas isso não o torna um chef. Da mesma forma, qualquer um pode digitar (ou copiar e colar) códigos, mas isso não o torna um programador.

Gastronomia, pintura, música, programação, são todas profissões de prática. A prática nos faz melhores em nossa arte. Não é algo que podemos meramente decorar e aplicar sem consciência do que significa ou aonde queremos chegar. E em qualquer profissão de prática existem fundamentos, conhecimentos essenciais, técnicas básicas, que quanto antes entendermos mais vão facilitar nossa evolução.

O Guilherme Silveira é um dos melhores programadores que já conheci, ele realmente tem não só a vocação, mas o talento tanto para assimilar esse tipo de conhecimento como para explicá-lo. É muito raro encontrar essa combinação.

Este livro que ele escreveu não vai torná-lo o grande "ninja" da programação. Não, ele vai lhe dar esse conhecimento essencial e as técnicas básicas que, se devidamente estudados, devem fornecer o alicerce para que quaisquer novas tecnologias do presente e do futuro que surjam sejam muito mais fáceis de assimilar.

Melhor ainda porque o Guilherme escolheu nossa amada linguagem Ruby para ajudá-los a aprender em uma plataforma que foi criada com conceitos

como estética e produtividade em mente. Através da linguagem Ruby e do contexto de criar pequenas lógicas de jogos, conceitos importantes como entrada e saída, funções, recursão, são explicados com clareza. Conceitos esses que são universais e importantes, não importa qual linguagem você usa hoje ou possa usar no futuro.

Bom estudo!

Capítulo 3

Jogo da adivinhação

3.1 O jogo: entrada e saída básica

Nosso primeiro grande projeto será a criação de um jogo que escolhe um número aleatório, e nos desafia a adivinhá-lo. O jogo permite a escolha do nível de dificuldade, nos dá feedback constante sobre nossos erros e acertos.

Portanto, nosso primeiro programa nos diz se o número é maior ou menor do que o escolhido pelo computador. Será um arquivo escrito na linguagem Ruby, que nome escolher para ele? Como o jogo brinca com maior ou menor, vamos chamar de `maior_ou_menor`, mas qual a extensão para ele? O padrão do Ruby é utilizar a extensão `rb`, portanto criamos um arquivo chamado `maior_ou_menor.rb`. O conteúdo? Vazio.

Dado esse programa, vazio, queremos mandar o computador executá-lo. Para isso dizemos para o `ruby` rodar nosso arquivo:

```
ruby maior_ou_menor.rb
```

E nada acontece. Claro, não havia nada para ser feito.

Um programa é uma lista de comandos que o computador obedece. Somos os donos do computador não só fisicamente, mas somos o dono de suas "mentes". Somos capazes de dizer comandos que o computador deve obedecer. Esses comandos serão escritos aqui na linguagem Ruby, e um tradutor, um intérprete, será capaz de traduzir esses comandos para um código "maluco" que nossa máquina entende. Afinal, quem hoje em dia quer aprender a falar a língua das máquinas? Muito complicado, principalmente para começar a programar. Então aprendemos uma linguagem mais próxima de nós do que dos computadores (*uma linguagem de alto nível*). Qual o primeiro comando que quero dar para meu computador? Por favor, imprima uma mensagem de boas-vindas.

Para isso dizemos para o computador colocar uma mensagem na saída: coloque, `puts`, a mensagem `"Bem-vindo ao jogo da adivinhação"`:

```
puts "Bem-vindo ao jogo da adivinhação"
```

A saída ao rodar novamente `ruby maior_ou_menor.rb` é agora o que esperamos:

```
Bem-vindo ao jogo da adivinhação
```

Já somos os donos. Ele nos obedece, e a mágica agora está em aprender mais comandos e formas de juntar comandos para criar programas complexos que fazem as mais diversas tarefas, desde um jogo até mesmo um piloto automático de avião. Todos eles são códigos escritos por seres humanos, e o computador obedece.

No nosso caso, queremos perguntar ao usuário qual é o seu nome, para personalizarmos sua experiência. Não é à toa que os jogos de hoje em dia perguntam o nome, no fim é possível lembrar quem foi o melhor jogador, criar um rank etc. No nosso caso começaremos pedindo o nome, usando novamente o `puts` para colocar uma mensagem na saída do computador:

```
puts "Bem-vindo ao jogo da adivinhação"
puts "Qual é o seu nome?"
```

E o inverso? Queremos que nosso programa leia um dado, pegue (`get`) informação do usuário, a entrada de dados mais simples é feita com a função `gets`, que devolve um texto, um valor que o usuário digitou, junto com o *enter* (o *return*):

```
nome = gets
```

Imprimimos o nome da mesma maneira que imprimimos outras mensagem:

```
puts "Bem-vindo ao jogo da adivinhação"
puts "Qual é o seu nome?"

nome = gets
puts "Começaremos o jogo para você, "
puts nome
```

E após executarmos com `ruby maior_ou_menor.rb` temos:

```
Bem-vindo ao jogo da adivinhação
Qual é o seu nome?
Guilherme
Começaremos o jogo para você,
Guilherme
```

Que saída horrível. Vamos separar o momento em que o nome foi lido do resto do programa, quando notificamos o usuário que começaremos o jogo. Primeiro colocamos algumas linhas em branco, isto é, não imprimimos nada, somente uma quebra de linha:

```
puts "Bem vindo ao jogo da adivinhação"
puts "Qual é o seu nome?"
nome = gets
puts
puts
puts
puts
puts
puts
puts "Começaremos o jogo para você, "
puts nome
```

E a saída agora fica um pouco melhor:

```
Bem-vindo ao jogo da adivinhação
Qual é o seu nome?
Guilherme






Começaremos o jogo para você,
Guilherme
```

A saída ainda está feia, gostaríamos de jogar na tela a mensagem `Começaremos o jogo para você, Guilherme`. Para isso precisamos juntar dois textos, o primeiro é o `nome`. Como fazer isso? Em linguagens de programação em geral, chamamos de *String* um valor que é um conjunto de caracteres, como uma palavra, um texto, uma placa de carro etc. Portanto, queremos juntar duas *String*, uma depois da outra. Como fazer isso? Usaremos a soma de duas `Strings`, chamada de *concatenação*:

```
puts "Começaremos o jogo para você, " + nome
```

Ficando com o código final:

```
puts "Bem-vindo ao jogo da adivinhação"
puts "Qual é o seu nome?"
nome = gets
puts
puts
puts
puts
puts
puts
puts "Começaremos o jogo para você, " + nome
```

Agora sim nossa saída é bonita:

```
Bem-vindo ao jogo da adivinhação
Qual é o seu nome?
Guilherme




Começaremos o jogo para você, Guilherme
```

Desejamos escolher um número secreto. Neste instante deixaremos um número fixo e depois o alteraremos para que cada vez o número tenha um valor diferente. Primeiro, imprimimos a mensagem de anúncio do sorteio, algo que já conhecemos:

```
puts "Escolhendo um número secreto entre 0 e 200..."
```

Depois queremos definir um novo valor. O valor `175` será nosso número secreto, logo falo que `numero_secreto` deve receber o valor `175`:

```
puts "Escolhendo um número secreto entre 0 e 200..."
numero_secreto = 175
puts "Escolhido... que tal adivinhar hoje nosso número secreto?"
```

Ficamos com o resultado:

```
Bem-vindo ao jogo da adivinhação
Qual é o seu nome?
Guilherme




Começaremos o jogo para você, Guilherme
Escolhendo um número secreto entre 0 e 200...
Escolhido... que tal adivinhar hoje nosso número secreto?
```

Precisamos agora perguntar um número, um chute, que o usuário deseja escolher. Já sabemos ler essa informação, portanto fazemos:

```
puts
puts
puts
puts
puts "Tentativa 1"
puts "Entre com o numero"
chute = gets
```

Mas o que fazer com o chute? Primeiro avisamos o usuário que processaremos seu chute, será que ele acertou? Novamente, é o código com que estamos começando a nos acostumar: `puts` passando `Strings` e concatenações:

```
puts
puts
puts
puts
puts "Tentativa 1"
puts "Entre com o numero"
chute = gets
puts "Será que acertou? Você chutou " + chute
```

Testamos nosso programa, chutando o número 100:

```
Bem-vindo ao jogo da adivinhação
Qual é o seu nome?
Guilherme








Começaremos o jogo para você, Guilherme
Escolhendo um número secreto entre 0 e 200...
Escolhido... que tal adivinhar hoje nosso número secreto?
```

```
Tentativa 1
Entre com o numero
100
Será que acertou? Você chutou 100
```

3.2 Será que acertou? O operador de comparação ==

Estamos prontos para verificar se o usuário acertou ou errou. Como? Queremos verificar se o valor chutado é igual ao número secreto. Qual o símbolo matemático para igualdade? O igual, `=`. Mas já usamos o `=` antes para dizer que um valor estava sendo utilizado. Como então ::verificar se um valor é igual a outro `? Usaremos o == . Será que o` chute

Por exemplo, será que 175 é igual a 175?

```
puts 175 == 175
```

Levando em consideração que verdadeiro é *true* em inglês, a saída é:

```
true
```

E será que ele é igual a 174?

```
puts 175 == 175
puts 175 == 174
```

Levando em consideração que falso é *false* em inglês, a saída é:

```
true
false
```

Isto é, o operador `==` realmente compara a igualdade entre dois números. Os valores verdadeiro e falso são chamados de *booleanos*.

Agora podemos verificar se o número chutado é igual a `175`:

```
puts 175 == chute
```

Ficando com o programa:

```
puts "Bem-vindo ao jogo da adivinhação"
puts "Qual é o seu nome?"
nome = gets
puts
puts
puts
puts
puts
puts
puts "Começaremos o jogo para você, " + nome
puts "Escolhendo um número secreto entre 0 e 200..."
numero_secreto = 175
puts "Escolhido... que tal adivinhar hoje nosso número secreto?"
puts
puts
puts
puts
puts "Tentativa 1"
puts "Entre com o numero"
chute = gets
puts "Será que acertou? Você chutou " + chute
puts 175 == chute
```

E eu errei:

```
...
Tentativa 1
Entre com o numero
100
Será que acertou? Você chutou 100
false
```

Mas e se eu chutar `175`?

```
...
Tentativa 1
Entre com o numero
175
```

```
Será que acertou? Você chutou 175
false
```

O que aconteceu? `175` é igual a `175`, como já sabíamos, mas ele imprimiu `false`. Acontece que o número `175` é igual ao número `175`, verdade. Mas lembra que o texto que o usuário entrou como informação, como seu chute, é um texto? O texto `"175"` não é "igual igual" ao número `175`. São duas coisas totalmente diferentes. Um é um texto, outro é um número. Antes mesmo de analisar seus valores, eles são duas coisas de tipos diferentes, seus tipos são diferentes: um é uma `String`, o outro é um `Int` (número inteiro).

```
puts "175" == 175
puts "175" == "175"
puts 175 == 175
```

Com o resultado:

```
false
true
true
```

Queremos então converter nosso chute para um número inteiro (*to an integer*), e adivinha? Existe um método chamado `to_i` que converte a `String` para um inteiro:

```
puts "175".to_i == 175
```

Resultando em:

```
true
```

Mas na verdade 175 é nosso `numero_secreto`, e queremos compará-lo com o `chute` portanto:

```
puts chute.to_i == numero_secreto
```

Terminando a primeira parte de nosso jogo com o código:

```
puts "Bem-vindo ao jogo da adivinhação"
puts "Qual é o seu nome?"
nome = gets
puts
puts
puts
puts
puts
puts
puts "Começaremos o jogo para você, " + nome
puts "Escolhendo um número secreto entre 0 e 200..."
numero_secreto = 175
puts "Escolhido... que tal adivinhar hoje nosso número secreto?"
puts
puts
puts
puts
puts "Tentativa 1"
puts "Entre com o numero"
chute = gets
puts "Será que acertou? Você chutou " + chute
puts 175 == chute.to_i
```

Agora sim nosso jogo de uma tentativa funciona, ou você acerta, ou você erra. Antes de continuarmos com as funcionalidades do jogo retomamos o que já vimos e escrevemos para entender mais a fundo os conceitos e melhorar nosso código.

3.3 Operador de atribuição e variáveis

Vimos que podemos utilizar o `=` para atribuir um valor a algo. Por exemplo, utilizamos `nome = gets` para ler do usuário um texto. A esse algo damos o nome de variável. No nosso caso, a variável `nome` recebe o valor que foi lido do jogador.

No nosso jogo temos três variáveis: o chute, o número secreto e o nome do jogador. Cada variável possui um valor. O número escolhido está fixo como `175`, enquanto tanto o nome do jogador quanto o chute são valores lidos a partir da entrada do usuário.

3.4 Operadores de comparação

Como o símbolo = foi utilizado pela linguagem para atribuir um valor a uma variável, teremos que adotar algum outro símbolo para a comparação de valores. O == é o padrão mais comum adotado pelas linguagens de programação para tal situação. Como vimos antes, o código a seguir imprime verdadeiro e depois falso:

```
puts 175 == 175
puts 175 == 174
```

Mas será que existem outros comparadores? Outro muito comum é o diferente, isto é, não (!) igual (=), como no exemplo a seguir, que imprime verdadeiro duas vezes:

```
puts 175 != 375
puts 175 != 174
```

Outros operadores utilizados comumente em operações matemáticas são os de maior (>), menor (<), maior ou igual (>=) ou menor ou igual (<=). Os exemplos a seguir todos imprimem verdadeiro:

```
puts 175 > 174
puts 174 < 175
puts 170 >= 170
puts 175 >= 170
puts 170 <= 170
puts 170 <= 175
```

3.5 Entrada e saída

Aprendemos a ler do teclado (entrada, *input*) invocando o `gets`, e a imprimir resultado na saída com o `puts`. Entrada e saída são vitais para um programa pois são a maneira padrão de ele se comunicar com o mundo exterior.

Por enquanto, a leitura é feita do teclado (na verdade, da entrada padrão), e a saída para o console (na verdade, a saída padrão). No decorrer deste material veremos como ler de e escrever para um arquivo, por exemplo.

No futuro é possível, por exemplo, ler da ou enviar informações para a internet, ler dados de um joystick ou se comunicar com uma caixa de som via bluetooth. Tudo isso são entrada e saída: o joystick é uma entrada de dados, uma caixa de som é uma saída.

3.6 O que são funções

Ao lermos e escrevermos dados utilizamos duas palavras importantes: `gets` e `puts`. Essas duas palavras não são meras palavras, são algo que podemos chamar, invocar, pedir para ser executado. São funções.

Uma função pode receber parâmetros, como no caso de `puts`:

```
puts "Bem-vindo ao jogo da adivinhação"
```

Em Ruby, ao invocar uma função, o uso dos parênteses é opcional em muitas situações, o caso a seguir mostra o mesmo código agora com parênteses:

```
puts("Bem-vindo ao jogo da adivinhação")
```

Desenvolvedores Ruby costumam invocar funções sem o uso dos parênteses, que é o padrão que seguiremos sempre que ficar claro o que está acontecendo. Assim como na matemática, o parêntese serve opcionalmente para deixar claro o que está acontecendo antes do quê.

Vimos também que em Ruby é possível ter uma função com parâmetro opcional, que é o próprio caso de `puts`. Os exemplos a seguir têm todos o mesmo resultado: a impressão de uma linha em branco.

```
puts ""
puts
puts("")
puts()
```

Por fim, uma função retorna alguma coisa. Lembra da função de soma na matemática? O que ela retornava? A soma de dois números. E a função de multiplicação? Um vezes o outro. O que a função de potência retornava? Um número multiplicado diversas vezes por ele mesmo? Toda função retorna

algo em Ruby e, no caso de `gets` ele retorna uma linha de entrada do usuário. Podemos fazer o que quisermos com esse retorno, como atribuí-lo a uma variável:

```
puts "Bem-vindo ao jogo da adivinhação"
puts "Qual é o seu nome?"
nome = gets
puts "Começaremos o jogo para você, " + nome
```

Mas o retorno de uma função já pode ser usado direto para invocar outra função:

```
puts "Bem-vindo ao jogo da adivinhação"
puts "Qual é o seu nome?"
puts "Começaremos o jogo para você, " + gets
```

Código estranho, né? Temos que tomar cuidado com código estranho. Se está estranho para nós, para quem ler este código daqui dois meses estará mais estranho ainda. Por isso mesmo não o escrevemos. Nosso objetivo como programadores não é escrever um código indecifrável, é escrever um código que funcione e que possa ser alterado no futuro sem criar bugs. É isso que buscaremos no decorrer deste material, aprender juntos a programar e escrever bons programas.

3.7 Refatoração: \n

Se nosso objetivo é escrever código melhor, pare aqui, Guilherme. Você escreveu seis `puts` só para pular seis linhas, poxa. Verdade. É hora de parar, olhar para trás e melhorar nosso código, o processo que chamamos de refatorar. Queremos alterar nossas seis linhas de `puts` para um código mais simples que alcance o mesmo resultado.

O que queremos é imprimir seis linhas em branco? Seria muito legal se fosse possível então escrever:

```
puts "proxima_linha,proxima_linha,proxima_linha,proxima_linha,
      proxima_linha,proxima_linha"
```

Claro, Ruby não entende isso. Mas existe uma sequência mágica (não, não é magia) que significa uma `nova` linha, *new line*, a sequência `\n`. Portanto, podemos substituir nosso código por:

```
puts "\n\n\n\n\n\n"
```

O resultado é o mesmo. Se isso funciona, podemos substituir também o `puts` de quatro linhas por quatro `\n`s. Maravilha, nosso código agora fica mais simples:

```
puts "Bem-vindo ao jogo da adivinhação"
puts "Qual é o seu nome?"
nome = gets
puts "\n\n\n\n\n\n"
puts "Começaremos o jogo para você, " + nome
puts "Escolhendo um número secreto entre 0 e 200..."
numero_secreto = 175
puts "Escolhido... que tal adivinhar hoje nosso número secreto?"
puts "\n\n\n\n"
puts "Tentativa 1"
puts "Entre com o numero"
chute = gets
puts "Será que acertou? Você chutou " + chute
puts 175 == chute.to_i
```

Refatoramos nosso código.

3.8 Interpretador ou compilador

Cada arquitetura de computador e sistema operacional possui um determinado conjunto de comandos, uma linguagem, que ele entende. Alguns dos sistemas compartilham os mesmos comandos, mas mesmo assim são comandos muito básicos, que permitem basicamente operações matemáticas (simples e complexas), e teríamos que implementar todo um programa ou jogo com eles se desejássemos utilizar sua linguagem. Não só isso, é necessário escrever para cada uma das arquiteturas de máquina e sistema operacional, um trabalho enorme. Escrever nessas linguagens mais baixo nível, mais próximas da linguagem da máquina (código de máquina) é, portanto, raro.

Na prática existe um mercado muito maior para desenvolvedores que escrevem em uma linguagem de mais alto nível, com menos preocupações tão pequenas. Nesse mundo, temos dois caminhos famosos. Podemos escrever nosso programa em forma de texto, como viemos fazendo até agora, e rodar outro programa que lê o nosso e traduz, interpreta, em tempo real para que o computador execute os comandos que desejamos.

Nesse sentido, temos que instalar primeiro esse interpretador, sendo que existe uma versão dele para cada arquitetura e sistema operacional. Ao mesmo tempo somente existirá uma versão do nosso código. Esse é o processo de interpretação, que a implementação padrão da linguagem Ruby utiliza.

Ou podemos pedir para um programa transformar o nosso texto, escrito uma única vez, na linguagem da máquina, o que geraria um programa autossuficiente, executável. Ele compila um programa para cada configuração de máquina desejada. Portanto essa outra abordagem, de um compilador, acaba gerando diversos arquivos executáveis. A linguagem C é famosa por ser compilada.

Existem também linguagens híbridas e diversas outras variações. A linguagem mais próxima e famosa de código de máquina, mas que não é código de máquina, é o *Assembly Language*.

A linguagem Java é um exemplo de variação, é uma linguagem primeiro compilada (arquivos `.class`), depois interpretada (*java virtual machine*) e, por fim, compilada para código de máquina (*just in time developer*).

O JRuby, um interpretador do Ruby baseado em Java, também se baseia na mesma abordagem híbrida.

3.9 Resumindo

Vimos até agora como criar um programa em Ruby, lidar com entrada e saída de variáveis simples como números inteiros e `String`s, comparar e converter números inteiros, além de usarmos algumas funções, invocar funções, entender como funcionam operadores básicos de comparação, e demos nosso primeiro passo em direção a um código melhor, entendendo o que é a refatoração, o processo de constante melhoria que permeia o dia a dia de um

desenvolvedor, sempre de uma maneira a não atrapalhar sua produtividade e qualidade. Vimos também como funcionam os processos de interpretação e de compilação de um programa.

Capítulo 4

Controle de fluxo

4.1 Mas e se... if e else

Escrevemos o início de nosso jogo, onde o jogador é capaz de chutar uma única vez o número entre 0 e 200 e dizemos se ele acertou ou não. Um jogo basicamente impossível de se ganhar. Mais ainda se a única informação que damos é `true` ou `false`, que horror. Primeiro vamos mudar essa mensagem para dizer algo como `Acertou!` ou `Errou!`.

Isto é, trocamos a linha do `puts 175 == chute.to_i` por duas linhas que imprimem essas mensagens:

```
puts "Bem-vindo ao jogo da adivinhação"
puts "Qual é o seu nome?"
nome = gets
puts "\n\n\n\n\n\n"
puts "Começaremos o jogo para você, " + nome
```

```
puts "Escolhendo um número secreto entre 0 e 200..."
numero_secreto = 175
puts "Escolhido... que tal adivinhar hoje nosso número secreto?"
puts "\n\n\n\n"
puts "Tentativa 1"
puts "Entre com o numero"
chute = gets
puts "Será que acertou? Você chutou " + chute

puts "Acertou!"
puts "Errou!"
```

Mas agora o programa não faz mais sentido nenhum: ele sempre imprime as duas mensagens! Queremos imprimir a primeira mensagem somente caso o nosso `numero_secreto` seja igual ao valor chutado (`chute.to_i`). E só vamos imprimir a mensagem de erro do chute caso contrário. Gostaríamos de fazer algo como:

```
se numero_secreto == chute.to_i
    puts "Acertou!"
se não
    puts "Errou!"
fim
```

Infelizmente, o Ruby não tem as palavras *se* e *se não* como palavra-chave para fazer esse código funcionar. Mas ele tem algo muito parecido, o `if`, `else`. Eles dois, junto com o `end`, permitem definir dois blocos de código que serão executados somente se a condição verificada for verdadeira (`if`) ou falsa (`else`), portanto se alterarmos nosso programa para:

```
if numero_secreto == chute.to_i
    puts "Acertou!"
else
    puts "Errou!"
end
```

Rodamos nosso jogo e temos o resultado adequado quando erramos:

```
...
Tentativa 1
```

```
Entre com o numero
160
Será que acertou? Você chutou 160
Errou!
```

If sem else: será?

É possível usar o `if` sem o `else` também. A segunda condição é opcional, como em:

```
if numero_secreto == chute.to_i
    puts "Acertou!"
end
```

Sempre que temos um `if` sem `else`, vale a pena nos perguntarmos o que acontecerá no caso contrário. Queremos mesmo ignorar a outra condição? Em nosso exemplo, não, e em muitos exemplos do dia a dia desejamos mostrar alguma mensagem de erro. Mas existem situações em que é natural usarmos somente a condição `if`. É importante sempre pensar e pesar qual abordagem se deseja utilizar. O exemplo a seguir mostra uma situação em que não utilizaríamos o `else` sem problema algum:

```
if numero_foi_muito_proximo
    put "Quase lá! Está chegando bem perto!"
end
```

Só que nosso código está meio feio, essa linha do `if` está fazendo bastante coisa, comparando um número com o valor de uma string transformada em inteiro. Quanta coisa! Para deixar claro o que está acontecendo, vamos comentar uma linha de nosso código: um comentário é um texto qualquer que será ignorado pelo interpretador do Ruby:

```
# Acertei ou não?
if numero_secreto == chute.to_i
    puts "Acertou!"
```

```
else
    puts "Errou!"
end
```

4.2 Code smell: comentários

Mas cuidado: comentários são indícios de que nosso código está difícil de entender. Nesse instante nosso código é até razoável, mas se existe a necessidade de comentá-lo pois não o entendemos direito, é hora de fazer melhorias. É hora de refatorá-lo. Já vimos um tipo de refatoração antes, agora veremos um novo. Temos uma condição `numero_secreto == chute.to_i`. O que ela nos diz? Se acertei ou não. Portanto, que tal extrairmos daí o seu valor em uma nova variável, *extract variable*:

```
acertou = numero_secreto == chute.to_i

if acertou
    puts "Acertou!"
else
    puts "Errou!"
end
```

Olha como nosso `if` ficou muito mais claro. Olhando agora o código refatorado, fica mais claro que o nosso condicional deixava as coisas um pouco mais complexas. No fim, temos o texto que diz exatamente quando o código será executado... "if acertou", isto é, se acertou, imprime `Acertou!`.

Outros tipos de comentários

Existem diversos tipos de comentários em Ruby. O mais famoso é o de uma linha:

```
# meu nome
nome = "Guilherme"
```

Ou o comentário de múltiplas linhas:

```
=begin
  Meus dados pessoais, começando pelo nome,
  passando até mesmo pela minha idade.
=end
nome = "Guilherme"
idade = 33
```

Mas o comentário de múltiplas linhas mais comum é o uso do próprio # em todas elas:

```
#  Meus dados pessoais, começando pelo nome,
#  passando até mesmo pela minha idade.
nome = "Guilherme"
idade = 33
```

Existem ainda outras maneiras de se comentar código em um arquivo, mas o mais importante é lembrar que, se precisa comentar, pode ser que precise refatorar. Repense seu código.

4.3 Condições aninhadas (nested ifs)

Mas o jogo ainda é muito difícil. Queremos deixá-lo mais fácil dizendo para o jogador, caso ele erre, se o número é maior ou menor do que o número chutado. Já conhecemos o operador de menor (<) e de maior (>), e as condições que podemos criar com `ifs`, portanto podemos verificar se o número é maior e dar a mensagem de acordo:

```
maior = numero_secreto > chute.to_i
if maior
    puts "O número secreto é maior!"
else
    puts "O número secreto é menor!"
end
```

Mas só queremos executar esse código caso o jogador não tenha acertado:

```
acertou = numero_secreto == chute.to_i

if acertou
    puts "Acertou!"
else
    maior = numero_secreto > chute.to_i
    if maior
        puts "O número secreto é maior!"
    else
        puts "O número secreto é menor!"
    end
end
```

O jogo funciona e a estrutura que criamos, com um `if` dentro de outro `if` é chamada de `nested if`. Repare que esse tipo de estrutura é um sinal de lógica mais complexa, afinal o código está fazendo diversas verificações, não só uma como fazia antigamente.

A variável diferença

Que tal criar uma variável chamada `diferenca`? Algo como:

```
diferenca = numero_secreto - chute.to_i
```

Poderíamos agora verificar se a diferença é zero (acertou), maior (o número secreto é maior) ou menor (o número secreto é menor) que zero. É uma abordagem válida mas ainda manteria duas ou três condições em nossos `ifs`, portanto não utilizaremos essa abordagem neste instante.

4.4 Code smell: copy e paste

Mesmo assim, o jogo continua muito difícil. Falta um último passo para permitir que nosso jogador tenha alguma chance de ganhar: temos que dar algumas chances, algumas vidas, para que ele possa tentar. Vamos dar 3 vidas, 3 tentativas, portanto fazemos um *copy e paste* do código da tentativa:

```
puts "Bem-vindo ao jogo da adivinhação"
puts "Qual é o seu nome?"
nome = gets
puts "\n\n\n\n\n\n"
puts "Começaremos o jogo para você, " + nome
puts "Escolhendo um número secreto entre 0 e 200..."
numero_secreto = 175
puts "Escolhido... que tal adivinhar hoje nosso número secreto?"



puts "\n\n\n\n"
puts "Tentativa 1"
puts "Entre com o numero"
chute = gets
puts "Será que acertou? Você chutou " + chute

acertou = numero_secreto == chute.to_i

if acertou
    puts "Acertou!"
else
    maior = numero_secreto > chute.to_i
    if maior
        puts "O número secreto é maior!"
    else
        puts "O número secreto é menor!"
    end
end
```

```
puts "\n\n\n\n"
puts "Tentativa 2"
puts "Entre com o numero"
chute = gets
puts "Será que acertou? Você chutou " + chute

acertou = numero_secreto == chute.to_i

if acertou
    puts "Acertou!"
else
    maior = numero_secreto > chute.to_i
    if maior
        puts "O número secreto é maior!"
    else
        puts "O número secreto é menor!"
    end
end



puts "\n\n\n\n"
puts "Tentativa 3"
puts "Entre com o numero"
chute = gets
puts "Será que acertou? Você chutou " + chute

acertou = numero_secreto == chute.to_i

if acertou
    puts "Acertou!"
else
    maior = numero_secreto > chute.to_i
    if maior
        puts "O número secreto é maior!"
    else
        puts "O número secreto é menor!"
    end
end
```

Agora sim, jogo e acerto na terceira tentativa:

```
...
Tentativa 1
Entre com o numero
100
Será que acertou? Você chutou 100
O número secreto é maior!
...
Tentativa 2
Entre com o numero
180
Será que acertou? Você chutou 180
O número secreto é menor!
...
Tentativa 3
Entre com o numero
175
Será que acertou? Você chutou 175
Acertou!
```

O jogo funciona e já tem uma certa graça, pois é desafiador acertar em três tentativas. Mas agora quero corrigir a mensagem de entrada do número, repare que esqueci de colocar acento em *número*. Que horror! Tenho que mudar o código em três lugares diferentes pois eu fiz *copy e paste*. Sim, veremos como melhorar esse código para dar andamento em nosso jogo!

4.5 O laço for (loop)

Nosso jogo atual pode divertir por alguns minutos mas é muito fácil perceber que será difícil de manter seu código à medida que o evoluímos. *Copy e paste* é uma das práticas mais nocivas a um código e abusamos dele aqui de propósito. Queremos executar o mesmo pedaço de código três vezes e para isso nos levei ao caminho do mal, do *copy e paste* sem pensar. É hora de refatorar.

Como seria bom poder dizer para o interpretador algo como:

```
execute 3 vezes
    puts "\n\n\n\n"
```

```
    puts "Tentativa 1"
    puts "Entre com o numero"
    chute = gets
    puts "Será que acertou? Você chutou " + chute

    acertou = numero_secreto == chute.to_i

    if acertou
        puts "Acertou!"
    else
        maior = numero_secreto > chute.to_i
        if maior
            puts "O número secreto é maior!"
        else
            puts "O número secreto é menor!"
        end
    end
end
```

Note que já adicionei um `end` para dizer para o interpretador onde nosso código a ser executado diversas vezes termina.

Mas a linguagem não sabe o que significa *3 vezes*. O que ela conhece são variáveis, números etc. Como representar 3 vezes em números? Da mesma maneira que uma criança representa os três números: 1, 2 e 3! Imagine Julieta, dois aninhos, para quem pedimos "por favor, conte até três". E ela conta: `1`, `2` e `3`. O que ela fez?

Ela criou uma variável chamada dedos, e colocou o número um lá dentro. Depois o número dois na variável dedos, depois o número três. Podemos mandar o computador fazer a mesma coisa, crie uma variável chamada `dedos` e execute o código a seguir para todos os números inteiros entre 1 e 3:

```
rode para dedos em 1 ate 3
    # código a ser executado
end
```

Estamos quase lá. Novamente, Ruby não fala português tão bem quanto a Julieta, portanto traduzimos para a linguagem dele, `1 até 3` vira `1..3`, enquanto `rode para` vira `for` e `em` vira `in`:

```
for dedos in 1..3
    # codigo a ser executado
end
```

Podemos testar com uma mensagem de impressão simples:

```
puts "Vou contar de 1 até 3"

for dedos in 1..3
    puts "Contando"
end
```

Resultando em:

```
Vou contar de 1 até 3
Contando
Contando
Contando
```

Só faltou contar de verdade, isto é, faltou imprimirmos o número que estamos contando. Dentro do código a ser executado diversas vezes temos acesso à variável chamada `dedos`, que é o valor que estamos passando agora:

```
puts "Vou contar de 1 até 3"

for dedos in 1..3
    puts "Contando: " + dedos
end
```

Mas ao rodar, percebemos que temos um erro de execução, algo ligado a não ser possível converter um número em uma `String`:

```
TypeError: no implicit conversion of Fixnum into String
```

Claro, não faz sentido somar um texto com um número. Faz sentido concatenar duas `Strings`, ou somar dois números. Mas um número e uma `String` não podem ser concatenados nem somados em Ruby. Vamos converter nosso número inteiro em uma `String` invocando seu método que transforma em `String`, o `to_s`:

```
puts "Vou contar de 1 até 3"

for dedos in 1..3
    puts "Contando: " + dedos.to_s
end
```

Resultando em:

```
Vou contar de 1 até 3
Contando: 1
Contando: 2
Contando: 3
```

4.6 Aplicando o laço *for* ao nosso programa

Pronto! O que temos aqui é um trecho de código que será repetido diversas vezes. Um *loop*, um *laço* que executa nosso código três vezes. O *for* é um dos diversos tipos de *loop* que podemos fazer e ainda veremos outras maneiras mais à frente.

Agora queremos trazer nosso laço para o código em que fizemos *copy e paste*. Queremos nos livrar da sujeira, executando o mesmo trecho três vezes. Para isso criamos um laço cuja variável `tentativa` vale entre `1` e `3`:

```
puts "Bem-vindo ao jogo da adivinhação"
puts "Qual é o seu nome?"
nome = gets
puts "\n\n\n\n\n\n"
puts "Começaremos o jogo para você, " + nome
puts "Escolhendo um número secreto entre 0 e 200..."
numero_secreto = 175
puts "Escolhido... que tal adivinhar hoje nosso número secreto?"


for tentativa in 1..3
    puts "\n\n\n\n"
    puts "Tentativa 1"
    puts "Entre com o numero"
    chute = gets
```

```
    puts "Será que acertou? Você chutou " + chute

    acertou = numero_secreto == chute.to_i

    if acertou
        puts "Acertou!"
    else
        maior = numero_secreto > chute.to_i
        if maior
            puts "O número secreto é maior!"
        else
            puts "O número secreto é menor!"
        end
    end
end
```

Agora nosso programa roda três vezes. Mas sempre imprime `Tentativa 1`. Corrigimos para mostrar a tentativa certa junto com o total de tentativas:

```
# ...

for tentativa in 1..3
    puts "\n\n\n\n"
    puts "Tentativa " + tentativa.to_s + " de 3"
    puts "Entre com o numero"
    chute = gets
    puts "Será que acertou? Você chutou " + chute

    acertou = numero_secreto == chute.to_i

    if acertou
        puts "Acertou!"
    else
        maior = numero_secreto > chute.to_i
        if maior
            puts "O número secreto é maior!"
        else
            puts "O número secreto é menor!"
```

```
        end
    end
end
```

Nosso jogo continua funcionando como antes, mas já sabemos que é bem mais fácil mantê-lo agora. Podemos corrigir a mensagem do `numero` sem acento trocando um único ponto do programa, pois não temos mais o *copy e paste*

```
puts "Entre com o número"
```

Mas 3 tentativas parece ser muito pouco para acertar. Queremos mudar para 5. Alteramos no nosso laço:

```
for tentativa in 1..5
```

E rodamos. Por mais que o jogo me dê cinco tentativas, ele imprime:

```
...
Tentativa 1 de 3
...
Tentativa 2 de 3
...
Tentativa 3 de 3
...
Tentativa 4 de 3
...
Tentativa 5 de 3
...
```

Que horror. Onde errei?

4.7 Code smell: Magic numbers

Repare que o número 3 apareceu duas vezes em nosso código. Tanto no laço quanto na impressão da mensagem de qual era a tentativa atual. O que há de errado nisso? Primeiro, nosso número está duplicado. Antes vimos um exemplo de *copy e paste* de um grande trecho de código, e percebemos que mudar um ponto seria trabalhoso pois teríamos que mudar todos.

Mas repare que mesmo quando temos um simples número repetido em duas partes do meu programa é muito fácil esquecer que ele existe. Também seria inviável procurar todas as partes de todos os meus arquivos onde o número `3` aparece. Ele quer dizer muitas coisas, e justamente por isso ele é um número mágico jogado no meu programa: ninguém sabe o que ele é, e ele aparece jogado em diversos lugares.

4.8 Refatoração: extrair variável

O que faremos é dar um nome a esse número. Vamos criar uma variável que se chama `limite_de_tentativas`:

```
limite_de_tentativas = 3
```

Efetuamos a refatoração de *extract variable* e a extraímos. Agora nosso número não é mais mágico, ele é muito bem definido e está claro para todos o que significa.

Após extrair a variável, refatoramos quem acessava esse valor (no caso, mágico) e substituímos por uma referência à variável que descreve seu valor:

```
for tentativa in 1..limite_de_tentativas
    puts "\n\n\n\n"
    puts "Tentativa " + tentativa.to_s + " de " +
          limite_de_tentativas.to_s
```

Pronto! Agora basta mudar o limite para `5`:

```
# ...

limite_de_tentativas = 5
for tentativa in 1..limite_de_tentativas
    # ...
end
```

4.9 Quebrando o laço com o break

Ótimo! Para terminarmos nossa primeira versão do jogo, temos que corrigir um único bug, nosso primeiro bug detectado pelo jogador. Se ele acertar de

primeira o número `175`, o jogo continua perguntando, feito bobo. Ainda não dá. Queremos alterar nosso jogo para, caso o usuário acerte, ele saia fora do laço, ele *quebre* (*break*) o nosso laço (*loop*).

Algo como:

```
if acertou
    puts "Acertou!"
    quebra
else
    maior = numero_secreto > chute.to_i
    if maior
        puts "O número secreto é maior!"
    else
        puts "O número secreto é menor!"
    end
end
```

Adivinha, o Ruby possui uma palavra-chave chamada `break` que quebra o laço atual, saindo dele completamente! Fazendo a alteração a seguir, o jogo termina lindamente mesmo que o jogador acerte de primeira:

```
puts "Bem-vindo ao jogo da adivinhação"
puts "Qual é o seu nome?"
nome = gets
puts "\n\n\n\n\n\n"
puts "Começaremos o jogo para você, " + nome
puts "Escolhendo um número secreto entre 0 e 200..."
numero_secreto = 175
puts "Escolhido... que tal adivinhar hoje nosso número secreto?"


limite_de_tentativas = 5
for tentativa in 1..limite_de_tentativas
    puts "\n\n\n\n"
    puts "Tentativa " + tentativa.to_s + " de " +
         limite_de_tentativas.to_s
    puts "Entre com o número"
    chute = gets
    puts "Será que acertou? Você chutou " + chute
```

```
    acertou = numero_secreto == chute.to_i

    if acertou
        puts "Acertou!"
        break
    else
        maior = numero_secreto > chute.to_i
        if maior
            puts "O número secreto é maior!"
        else
            puts "O número secreto é menor!"
        end
    end
end
```

Com o resultado:

```
...
Tentativa 1 de 5
Entre com o número
175
Será que acertou? Você chutou 175
Acertou!
```

4.10 Resumindo

Vimos neste capítulo como funciona o operador `+`, que é válido entre `Strings`, quando executa uma operação de concatenação, e entre números, ao executar uma soma. Aprendemos a converter um número inteiro em `String`, executar código condicionalmente com o uso do `if...else...end`, a utilização e o pepino de se usar `ifs` aninhados, como funciona e (novamente) o mal cheiro que um comentário indica, a refatoração de extrair variável, o controle de fluxo através de um laço, permitindo a execução repetida de um trecho de código, além de como quebrar esse laço a qualquer instante com o uso do `break`. Nosso jogo já nos desafia a acertar um número entre 0 e 200 (pode testar, é difícil alguém acertar em tão poucas

tentativas). Nosso próximo passo é melhorar bastante esse nosso código antes de adicionar diversas novas funcionalidades.

Capítulo 5

Funções

5.1 Funções

Somos capazes de jogar com um mínimo de diversão, mas antes de continuarmos com as funcionalidades avançadas, queremos ter certeza de que outro desenvolvedor será capaz de contribuir com nosso projeto.

No mundo moderno é muito comum que projetos sejam desenvolvidos por mais de um programador, seja dentro de uma pequena ou grande empresa, ou ainda nos casos de projetos de código aberto, nos quais, através de plataformas como o Github e tecnologias como o Git, desenvolvedores do mundo inteiro contribuem com os mais variados projetos, desde o core do linux até o interpretador padrão da linguagem Ruby em si.

Sendo assim, vamos dar uma olhada novamente em nosso código. O que mais me chama a atenção é que temos um único arquivo com mais de trinta

linhas e que faz tudo lá mesmo, sem rédeas, nem limites. Infelizmente é comum encontrar código como esse no mundo selvagem, mas para nós, bons programadores, tudo (ou muita coisa) em um único lugar é um *code smell* que queremos aos poucos combater.

Comecemos pelo início do código, onde pedimos o nome do usuário:

```
puts "Bem-vindo ao jogo da adivinhação"
puts "Qual é o seu nome?"
nome = gets
puts "\n\n\n\n\n\n"
puts "Começaremos o jogo para você, " + nome
puts "Escolhendo um número secreto entre 0 e 200..."
numero_secreto = 175
puts "Escolhido... que tal adivinhar hoje nosso número secreto?"


limite_de_tentativas = 5
# ...
```

Nesse trecho de código pedimos tal informação e imprimimos a mensagem de boas-vindas, isso é, damos início ao jogo como um todo:

```
puts "Bem-vindo ao jogo da adivinhação"
puts "Qual é o seu nome?"
nome = gets
puts "\n\n\n\n\n\n"
puts "Começaremos o jogo para você, " + nome
```

Mas um novo desenvolvedor tem que olhar esse código linha a linha para entender o que ele está fazendo. De que maneira poderíamos deixar isso mais claro? Com um comentário, como em:

```
# dá boas-vindas
puts "Bem-vindo ao jogo da adivinhação"
puts "Qual é o seu nome?"
nome = gets
puts "\n\n\n\n\n\n"
puts "Começaremos o jogo para você, " + nome
puts "Escolhendo um número secreto entre 0 e 200..."
```

```
numero_secreto = 175
puts "Escolhido... que tal adivinhar hoje nosso número secreto?"


limite_de_tentativas = 5
# ...
```

Mas já vimos anteriormente que um comentário é um cheiro fétido no código (*code smell*), o que fazer? Queremos extrair esse código. Diferentemente do que fizemos antes, um pedaço de código não é um mero valor, como um número ou uma `String`, ele é algo a ser executado, e essa diferença é importante. Tanto um pedaço de código, um comportamento, pode ter um nome quanto um valor pode ter um nome. Mas somente um pedaço de código pode ser executado, invocado.

Qual a maneira padrão de extrair um código para poder invocá-lo? Primeiro, substituímos nosso código pelo que desejamos escrever, chamar, invocar:

```
da_boas_vindas
puts "Escolhendo um número secreto entre 0 e 200..."
numero_secreto = 175
puts "Escolhido... que tal adivinhar hoje nosso número secreto?"


limite_de_tentativas = 5
# ...
```

Agora precisamos definir (*define*, abreviado *def*) o significado de *da_boas_vindas*, portanto:

```
def da_boas_vindas

end
```

E dizemos o que será executado quando alguém chama, invoca, o `da_boas_vindas`:

```
def da_boas_vindas
    puts "Bem-vindo ao jogo da adivinhação"
```

```
    puts "Qual é o seu nome?"
    nome = gets
    puts "\n\n\n\n\n\n"
    puts "Começaremos o jogo para você, " + nome
end
```

O que criamos? Uma função (*function*), que é um trecho de código que pode ser executado quantas vezes desejarmos, bastando para isso invocá-lo (ou chamá-lo) através de seu nome. Uma função é peça fundamental de reaproveitamento de código e será parte do seu dia a dia de programador. Em Ruby, a maneira padrão e usual de definir uma função é através da construção `def ... end` e invocá-la através de seu nome. Repare que a nomenclatura segue o padrão com *underline* para separar palavras.

5.2 Boa prática: encapsulamento de comportamento

Repare que agora um desenvolvedor novo é capaz de entender o que significa `da_boas_vindas` sem ter que olhar o código inteiro: o próprio nome diz tudo! Ele dá boas-vindas ao jogador. Ao chamar `da_boas_vindas` também não é necessário entender como é feito o esquema de boas-vindas, simplesmente basta chamar, dizer, invocar `da_boas_vindas`, sem se preocupar em como o comportamento é feito, mas somente com a interface, ou seja, com o que ele faz, não como.

```
def da_boas_vindas
    puts "Bem-vindo ao jogo da adivinhação"
    puts "Qual é o seu nome?"
    nome = gets
    puts "\n\n\n\n\n\n"
    puts "Começaremos o jogo para você, " + nome
end

da_boas_vindas
puts "Escolhendo um número secreto entre 0 e 200..."
numero_secreto = 175
puts "Escolhido... que tal adivinhar hoje nosso número secreto?"
```

```
# ...
```

Escondemos como o comportamento é executado, somente mostramos o que é feito através do nome da nossa função. Esse processo de esconder como as coisas são feitas, apenas deixando disponível uma interface que indica o que será feito e o que é necessário para que ele seja executado, é chamado de *encapsulamento*, algo muito bem recebido por bons programadores. Ele facilita a manutenção do nosso código em longo prazo ao permitir compreender mais facilmente trechos isolados, além de outras vantagens que veremos com o passar do tempo e tipos diferentes de encapsulamento.

5.3 Escopo de variáveis

Assim como extraímos o código referente às boas-vindas, desejamos fazer o mesmo com o código que sorteia o número secreto. Repetimos o processo de definir a função:

```
def sorteia_numero_secreto
    puts "Escolhendo um número secreto entre 0 e 200..."
    numero_secreto = 175
    puts "Escolhido... que tal adivinhar hoje nosso número
          secreto?"
end
```

E passamos a invocá-la:

```
da_boas_vindas
sorteia_numero_secreto

limite_de_tentativas = 5
# ...
```

O que acontece? O código para de funcionar! O número sorteado não é mais 175?

```
...
Tentativa 1 de 5
```

```
Entre com o número
175
Será que acertou? Você chutou 175
maior_ou_menor.rb:26:in `block in <main>':
        undefined local variable or method `numero_secreto'
        for main:Object (NameError)
    from maior_ou_menor.rb:19:in `each'
    from maior_ou_menor.rb:19:in `<main>'
```

Note que a mensagem de erro em tempo de execução do interpretador em Ruby nos diz exatamente qual erro aconteceu em qual linha de código. Lendo de baixo para cima, a linha 19 de nosso arquivo *maior_ou_menor.rb* executou um *each* (nosso `for`), quando de repente, na linha 26, o código tentou acessar uma variável local ou método indefinido, o `numero_secreto`. Como assim indefinido?

Antigamente havíamos utilizado uma variável que foi definida no programa como um todo, não em um escopo específico. Ela era definida logo de cara no programa, e todo o programa podia acessá-la, ela era aberta a todos. Agora não fazemos mais assim. Nossa variável é definida dentro de um bloco, no caso, dentro de nossa função:

```
def sorteia_numero_secreto
    puts "Escolhendo um número secreto entre 0 e 200..."
    numero_secreto = 175
    puts "Escolhido... que tal adivinhar hoje nosso número
          secreto?"
end
```

Uma variável definida dentro de uma função (por enquanto) só é visível dentro desse espaço, e esse espaço é chamado de escopo (*scope*) da variável. Variáveis desse tipo são chamadas variáveis locais, que são muito boas, pois nos permitem controlar quem pode e quem não pode acessá-las.

5.4 Code smell: variáveis sem controle de escopo e variáveis globais

Variáveis globais são o cúmulo, o máximo, do descontrole. O pensamento costuma ser assim: hoje acreditamos que só precisamos de uma dela. Se amanhã precisamos de duas, não dá mais, temos que alterar todo o nosso código. Se escrevemos uma biblioteca que outros desenvolvedores vão usar, minhas variáveis globais podem ter o mesmo nome que as deles, ou, pior ainda, se eu mudar o nome de minha variável (ou função, ou qualquer coisa) global, quebro a compatibilidade com todos os meus clientes. Como regra geral, quanto mais coisa global, pior.

São famosos e inúmeros os casos de quebra de compatibilidade entre versões ou de incompatibilidade entre bibliotecas devido ao abuso de elementos globais.

Em Ruby, a variável definida "voando" não é *global* ao pé da letra. Ela pode ser acessada por qualquer outro código "voando", que não está explicitamente dentro de nenhuma outra função, mas de qualquer maneira variáveis definidas assim têm seu controle de escopo, digamos, descontrolado. Na prática, ao utilizá-las não há controle de escopo. Evite-as.

Usando variáveis globais

Como referência, variáveis realmente globais (e aí também mora o perigo) são definidas com `$` na frente, como no exemplo a seguir, onde qualquer um pode acessar, com encapsulamento e controle zero:

```
$nome = "Guilherme"
```

No exemplo a seguir temos a variável tradicional declarada fora de qualquer função, "voando".

```
nome = "Guilherme"
```

Quando qualquer um "fora de uma função" pode acessar uma variável, o encapsulamento é quase zero, e o controle sobre seu código também.

5.5 Retorno de função

Mas então como fazer com que nossa função devolva o número secreto escolhido? Queremos que ela retorne dizendo algo, que retorne um valor. Se ela vai retornar um valor, `return` nela:

```
def sorteia_numero_secreto
    puts "Escolhendo um número secreto entre 0 e 200..."
    numero_secreto = 175
    puts "Escolhido... que tal adivinhar hoje nosso número
          secreto?"
    return numero_secreto
end
```

E se vamos invocá-la, podemos atribuir seu retorno a uma nova variável:

```
da_boas_vindas
numero_secreto = sorteia_numero_secreto

limite_de_tentativas = 5
for tentativa in 1..limite_de_tentativas
    puts "\n\n\n\n"
    puts "Tentativa " + tentativa.to_s + " de " +
         limite_de_tentativas.to_s
    puts "Entre com o número"
    chute = gets
    puts "Será que acertou? Você chutou " + chute

    acertou = numero_secreto == chute.to_i

    if acertou
        puts "Acertou!"
        break
    else
        maior = numero_secreto > chute.to_i
        if maior
            puts "O número secreto é maior!"
        else
            puts "O número secreto é menor!"
        end
```

```
    end
end
```

Pronto! Nosso código volta a funcionar. Mas fique atento: a variável local que existia durante a chamada da função `sorteia_numero_secreto` é diferente da que foi criada depois, apesar de terem o mesmo valor, claro. Podemos deixar isso claro mudando o nome da variável da função:

```
def sorteia_numero_secreto
    puts "Escolhendo um número secreto entre 0 e 200..."
    sorteado = 175
    puts "Escolhido... que tal adivinhar hoje nosso número
          secreto?"
    return sorteado
end
```

Note que em Ruby o retorno da expressão da última linha de código sempre é retornado por uma função. Por isso é muito comum que os programadores Ruby não escrevam a palavra `return` na última linha de uma função, ficando somente:

```
def sorteia_numero_secreto
    puts "Escolhendo um número secreto entre 0 e 200..."
    sorteado = 175
    puts "Escolhido... que tal adivinhar hoje nosso número
          secreto?"
    sorteado
end
```

5.6 Variáveis locais

Faltam agora mais dois passos para terminarmos de extrair todas as funções que queremos. Desejamos extrair o código de perguntar o número para nosso jogador, e o que verifica se ele ganhou. Começamos com o que coloca em prática o conhecimento de funções que adquirimos até agora: extrair a pergunta, que é o nosso código a seguir:

```
puts "\n\n\n\n"
puts "Tentativa " + tentativa.to_s + " de " +
```

```
    limite_de_tentativas.to_s
puts "Entre com o número"
chute = gets
puts "Será que acertou? Você chutou #{chute}"
```

Qual o nome que daremos a esse código? Como ele *pede um número* para o usuário, vamos chamá-lo de `pede_um_numero`. Já sabemos que o código precisará retornar o número que foi pedido, afinal ele será utilizado em breve para verificar se o jogador acertou ou errou, portanto extraímos o código junto com o retorno na função `pede_um_numero`:

```
def pede_um_numero
    puts "\n\n\n\n"
    puts "Tentativa " + tentativa.to_s + " de " +
         limite_de_tentativas.to_s
    puts "Entre com o número"
    chute = gets
    puts "Será que acertou? Você chutou " + chute
    chute
end
```

Agora nosso código do loop principal do jogo fica:

```
da_boas_vindas
numero_secreto = sorteia_numero_secreto

limite_de_tentativas = 5
for tentativa in 1..limite_de_tentativas
    chute = pede_um_numero

    acertou = numero_secreto == chute.to_i

    if acertou
        puts "Acertou!"
        break
    else
        maior = numero_secreto > chute.to_i
        if maior
            puts "O número secreto é maior!"
```

```
        else
            puts "O número secreto é menor!"
        end
    end
end
```

Mesmo assim ainda tem algo de estranho com o chute. Ele sempre é utilizado com o `.to_i`. Isto é, o loop principal é invejoso, ele quer sempre fazer algo que quem sabe fazer é o número, transformar uma `String` em número. Não queremos uma `String`, e nem a utilizamos, estamos sempre transformando em número. Até o nome da função diz isso: `pede_um_numero`, e estranhamente ela devolve uma `String`. Vamos corrigir essa possível fonte de bugs agora mesmo, retornando o número:

```
def pede_um_numero
    puts "\n\n\n\n"
    puts "Tentativa " + tentativa.to_s + " de " +
         limite_de_tentativas.to_s
    puts "Entre com o número"
    chute = gets
    puts "Será que acertou? Você chutou " + chute
    chute.to_i
end
```

E agora utilizar o número já educadamente:

```
da_boas_vindas
numero_secreto = sorteia_numero_secreto

limite_de_tentativas = 5
for tentativa in 1..limite_de_tentativas
    chute = pede_um_numero

    acertou = numero_secreto == chute

    if acertou
        puts "Acertou!"
        break
    else
```

```
        maior = numero_secreto > chute
        if maior
            puts "O número secreto é maior!"
        else
            puts "O número secreto é menor!"
        end
    end
end
```

Tentamos rodar a aplicação e recebemos um erro novo:

```
maior_ou_menor.rb:18:in `pede_um_numero':
    undefined local variable or method `tentativa'
    for main:Object (NameError)

    from maior_ou_menor.rb:30:in `block in <main>'
    from maior_ou_menor.rb:29:in `each'
    from maior_ou_menor.rb:29:in `<main>'
```

Já vimos algo parecido com *variável local ou método não definido* anteriormente. Tentávamos usar uma variável que só era visível em outro escopo. Será que é isso que ocorre aqui novamente? `undefined local variable` costuma ser o que ele mesmo diz: não existe algo com esse nome neste escopo.

Acompanhemos a linha onde ocorreu o erro: linha 29 efetuou o `for` (*each*) e quando chegou na linha 30, invocou a função `pede_um_numero` que chegou até a linha 18, onde não encontrou ninguém chamado `tentativa`. Minha linha 18 é onde tentamos acessar a `tentativa` em:

```
def pede_um_numero
    puts "\n\n\n\n"
    puts "Tentativa " + tentativa.to_s + " de " +
         limite_de_tentativas.to_s
    puts "Entre com o número"
    chute = gets
    puts "Será que acertou? Você chutou #{chute}"
    chute.to_i
end
```

Realmente, a variável não foi definida aqui dentro! Ela foi definida dentro do nosso laço `for`. Toda vez que começamos um novo escopo, o escopo

das variáveis é modificado, podendo ou não ter acesso às variáveis que estavam disponíveis antes da invocação. Por exemplo, quando efetuamos nosso `for`, continuamos podendo acessar as variáveis de fora do `for` normalmente. Mas ao invocarmos uma função definida como fizemos até aqui, não temos acesso às variáveis que foram declaradas antes da chamada da função. Isso quer dizer que temos um problema. Como acessar as variáveis `tentativa` e `limite_de_tentativas` de que precisamos?

Lembra do encapsulamento? É ele que nos dá o poder de acessar somente o que é necessário, evitando um código macarrônico, uma mistura que faz tudo de tudo que é jeito. Ao encapsularmos esse comportamento de pedir um número surgiu a necessidade de acessarmos dois valores, nada mais natural e justo que passemos esses dois valores para a função `pede_um_numero`, fazemos essa passagem através de dois argumentos, dois parâmetros:

```
def pede_um_numero(tentativa, limite_de_tentativas)
    puts "\n\n\n\n"
    puts "Tentativa " + tentativa.to_s + " de " +
         limite_de_tentativas.to_s
    puts "Entre com o número"
    chute = gets
    puts "Será que acertou? Você chutou #{chute}"
    chute.to_i
end
```

Invocamos a função passando os dois valores:

```
da_boas_vindas
numero_secreto = sorteia_numero_secreto

limite_de_tentativas = 5
for tentativa in 1..limite_de_tentativas
    chute = pede_um_numero(tentativa, limite_de_tentativas)

    # ...
end
```

Agora sim nosso programa volta a funcionar. Repare que as funções são um primeiro recurso básico de encapsulamento que nos permite esconder

o comportamento, recebendo como parâmetros o que é necessário e devolvendo como retorno o resultado da execução do código.

Propositalmente isolamos nosso código de maneira que as funções tivessem uma granularidade, um tamanho e uma quantidade de lógica pequenos. Funções grandes e complexas podem ser sempre quebradas em pedaços menores, possivelmente mais fáceis de serem compreendidos.

5.7 Extraindo mais uma função

Agora falta só mais um passo: extrair o código, a lógica de negócios e de saída para o usuário, que verifica se o jogador acertou:

```
acertou = numero_secreto == chute

if acertou
    puts "Acertou!"
    break
else
    maior = numero_secreto > chute
    if maior
        puts "O número secreto é maior!"
    else
        puts "O número secreto é menor!"
    end
end
```

O que esse código faz? Ele *verifica se acertou*, portanto criaremos uma função `verifica_se_acertou`. Note que ela precisa saber qual o número secreto e qual o chute do jogador, logo receberemos os dois como parâmetro:

```
def verifica_se_acertou(numero_secreto, chute)
    acertou = numero_secreto == chute
    if acertou
        puts "Acertou!"
        break
    else
        maior = numero_secreto > chute
        if maior
```

```
            puts "O número secreto é maior!"
        else
            puts "O número secreto é menor!"
        end
    end
end
```

Vamos invocá-la no nosso loop principal:

```
da_boas_vindas
numero_secreto = sorteia_numero_secreto

limite_de_tentativas = 5
for tentativa in 1..limite_de_tentativas
    chute = pede_um_numero(tentativa, limite_de_tentativas)
    verifica_se_acertou(numero_secreto, chute)
end
```

Tentamos rodar e o código nem começa a ser rodado. O Ruby parou, mas não foi um erro de execução. O interpretador não conseguiu nem ler nosso arquivo `rb`. Isso pois nossa estrutura está inválida agora, temos um erro de "compilação", um erro em nossa sintaxe:

```
maior_ou_menor.rb:29: Invalid break
maior_ou_menor.rb: compile error (SyntaxError)
```

Na linha 29 tentamos executar o *break* dentro de uma função, o que não tem problema. Mas lembre-se que o *break* quebra o loop mais próximo, e qual o loop que está ocorrendo naquela função? Nenhum. Entendo, você como ser humano sabe que um outro ponto do programa (nosso loop principal) invocará aquela função, que então terá um *break*. Mas o interpretador não tem como saber isso. Pior ainda, pode ser que um programador desavisado invoque tal função sem estar dentro de um loop. É por isso mesmo que a sintaxe da linguagem não permite um *break* solto, voltando. O *break* só pode existir se há um loop visível no escopo atual, que não é o caso. Mas então como falar para o nosso laço terminar?

Se a função verifica se acertou, parece natural ela dizer se sim ou se não. Fazemos:

```
def verifica_se_acertou(numero_secreto, chute)
    acertou = numero_secreto == chute
    if acertou
        puts "Acertou!"
        return true
    else
        maior = numero_secreto > chute
        if maior
            puts "O número secreto é maior!"
            return false
        else
            puts "O número secreto é menor!"
            return false
        end
    end
end
```

Agora ao invocar a função podemos quebrar nosso *loop*:

```
da_boas_vindas
numero_secreto = sorteia_numero_secreto

limite_de_tentativas = 5
for tentativa in 1..limite_de_tentativas
    chute = pede_um_numero(tentativa, limite_de_tentativas)
    if verifica_se_acertou(numero_secreto, chute)
        break
    end
end
```

Pronto! Nosso jogo voltou a funcionar. Uma última refatoração de boa prática de código envolve mudar nossos `ifs`:

```
def verifica_se_acertou(numero_secreto, chute)
    acertou = numero_secreto == chute
    if acertou
        puts "Acertou!"
        return true
    else
        maior = numero_secreto > chute
```

```
        if maior
            puts "O número secreto é maior!"
        else
            puts "O número secreto é menor!"
        end
        return false
    end
end
```

5.8 Boa prática: early return

Repare que, caso o usuário tenha acertado, nós saímos da função logo de cara. Isso se chama de saída *cedo* (*early return*). Se saímos da função, podemos jogar todo o código do erro para fora do `if`:

```
def verifica_se_acertou(numero_secreto, chute)
    acertou = numero_secreto == chute
    if acertou
        puts "Acertou!"
        return true
    end
    maior = numero_secreto > chute
    if maior
        puts "O número secreto é maior!"
    else
        puts "O número secreto é menor!"
    end
    return false
end
```

Como o retorno de falso está na última linha, podemos omitir a palavra `return`:

```
def verifica_se_acertou(numero_secreto, chute)
    acertou = numero_secreto == chute
    if acertou
        puts "Acertou!"
        return true
    end
```

```
    maior = numero_secreto > chute
    if maior
        puts "O número secreto é maior!"
    else
        puts "O número secreto é menor!"
    end
    false
end
```

5.9 Pequenas refatorações específicas da linguagem

Pronto, não há mais refatorações que gostaríamos de fazer nesse instante que são, em geral, universais e independentes de linguagem. Mas existe um aspecto da linguagem que os programadores Ruby em geral preferem. O primeiro envolve a invocação de funções. Lembra que ao invocar o `puts` o parênteses era opcional? No nosso caso também, portanto mudamos nosso código para:

```
da_boas_vindas
numero_secreto = sorteia_numero_secreto

limite_de_tentativas = 5
for tentativa in 1..limite_de_tentativas
    chute = pede_um_numero tentativa, limite_de_tentativas
    if verifica_se_acertou numero_secreto, chute
        break
    end
end
```

Outra característica da linguagem bastante utilizada por desenvolvedores Ruby envolve a nossa condição de quebra do laço. Quando temos somente uma instrução de código a ser executado em um `if` é comum que os programadores façam:

```
break if verifica_se_acertou numero_secreto, chute
```

Em vez de:

```
if verifica_se_acertou numero_secreto, chute
    break
end
```

Apesar de ser o padrão dos programadores Ruby, nesse caso específico deixarei a versão tradicional, com `if ... end`, que acredito ser mais compreensível para nós programadores nesse instante de nossa carreira.

5.10 Resumindo: o poder da extração de código

Nosso código final é composto agora por quatro funções independentes, que podem ser entendidas por qualquer desenvolvedor isoladamente. Seus nomes também dizem bastante sobre o que fazem. O código a seguir dá boas-vindas:

```
def da_boas_vindas
    puts "Bem vindo ao jogo da adivinhação"
    puts "Qual é o seu nome?"
    nome = gets
    puts "\n\n\n\n\n\n"
    puts "Começaremos o jogo para você, " + nome
end
```

Podemos a qualquer instante sortear um novo número secreto:

```
def sorteia_numero_secreto
    puts "Escolhendo um número secreto entre 0 e 200..."
    sorteado = 175
    puts "Escolhido... que tal adivinhar hoje nosso número
          secreto?"
    sorteado
end
```

Ou pedir um número para o jogador:

```
def pede_um_numero(tentativa, limite_de_tentativas)
    puts "\n\n\n\n"
    puts "Tentativa " + tentativa.to_s + " de " +
         limite_de_tentativas.to_s
    puts "Entre com o número"
```

```
    chute = gets
    puts "Será que acertou? Você chutou " + chute
    chute.to_i
end
```

Podemos também verificar se ele acertou ou errou o chute:

```
def verifica_se_acertou(numero_secreto, chute)
    acertou = numero_secreto == chute
    if acertou
        puts "Acertou!"
        return true
    end
    maior = numero_secreto > chute
    if maior
        puts "O número secreto é maior!"
    else
        puts "O número secreto é menor!"
    end
    false
end
```

Por fim, nosso código do laço principal é composto de todos esses comportamentos:

```
da_boas_vindas
numero_secreto = sorteia_numero_secreto

limite_de_tentativas = 5
for tentativa in 1..limite_de_tentativas
    chute = pede_um_numero tentativa, limite_de_tentativas
    if verifica_se_acertou numero_secreto, chute
        break
    end
end
```

5.11 Resumindo

Aprendemos como criar nossas funções, encapsulando parte do comportamento, da lógica de uma tarefa. Entendemos melhor como funciona o es-

copo de variáveis: aquelas definidas dentro de uma função são locais a ela. Vimos também que variáveis sem controle de escopo são as que definimos abertamente em nosso código e que podemos também criar variáveis globais, fáceis de serem acessadas mas perigosas para uma aplicação. Vimos como receber argumentos, parâmetros, e retornar valores de nossas funções. Por fim, aprendemos como funciona o `early return`, que permite sair mais cedo da função e, por vezes, simplificar nosso código.]

Capítulo 6

Arrays

Testei o jogo aqui com minha mãe, mas ela sem querer digitou duas vezes o mesmo número. E botou a culpa no programa outra maneira de dizer que o programador bobeou, claro. E bobeei mesmo, eu deveria tomar o cuidado e proteger o jogador de não tentar chutar o mesmo número duas vezes. Nesse instante ele precisa lembrar de cabeça todos os números que chutou, o que é muito difícil.

Como memorizar o último número chutado? Com uma variável, claro:

```
ultimo_chute = -1

limite_de_tentativas = 5
for tentativa in 1..limite_de_tentativas
    chute = pede_um_numero tentativa, limite_de_tentativas
    if verifica_se_acertou numero_secreto, chute
        break
```

```
    end

    ultimo_chute = chute
end
```

Mas e o penúltimo chute?

```
ultimo_chute = -1
penultimo_chute = -1

limite_de_tentativas = 5
for tentativa in 1..limite_de_tentativas
    chute = pede_um_numero tentativa, limite_de_tentativas
    if verifica_se_acertou numero_secreto, chute
        break
    end

    penultimo_chute = ultimo_chute
    ultimo_chute = chute
end
```

E o antepenúltimo chute? E o anteantepenúltimo chute? E o que veio antes desse? Daria um nó na minha cabeça de tanta variável. Não quero tudo isso jogado em uma dezena de variáveis. Pior ainda, se mudarmos o número limite de tentativas de 5 para outro número qualquer, teríamos que mudar também o número de variáveis, reescrever o código etc. Que trabalho.

Como sempre, vamos parar para pensar e fazer uma única pergunta: o que queremos? Uma maneira de guardar diversos números, de preferência agrupados para que eu não tenha cinquenta nomes de variáveis totalmente diferentes.

Que tal uma única variável capaz de guardar cinco números?

6.1 Criando e manipulando arrays

O que queremos aqui é armazenar uma lista de números. Uma sequência de números que poderemos referenciar através de uma variável só. Essa lista de números é um array em Ruby e veremos aqui algumas das coisas que um *array* é capaz de fazer.

Para criarmos um *array* com cinco números, usamos:

```
chutes = [176, 100, 130, 150, 175]
```

Se queremos acessar a primeira casinha, o primeiro número, utilizamos o número zero dentro de colchetes:

```
chutes = [176, 100, 130, 150, 175]
puts chutes[0] # imprime 176
```

Portanto, tome cuidado! Em Ruby, assim como em diversas outras linguagens, o padrão é começar no número zero. Se quisermos acessar o número `100`, usaremos:

```
chutes = [176, 100, 130, 150, 175]
puts chutes[1] # imprime 100
```

Ou ainda podemos imprimir o `175`:

```
chutes = [176, 100, 130, 150, 175]
puts chutes[4] # imprime 175
```

Para alterar o valor dentro de um array usamos o mesmo colchete e a contagem da posição começando com zero:

```
chutes = [176, 100, 130, 150, 175]
chutes[3] = 300

puts chutes[2] # imprime 100
puts chutes[3] # imprime 300
puts chutes[4] # imprime 150
```

Podemos também criar um array vazio e colocar valores aos poucos:

```
chutes = []
chutes[0] = 50
chutes[1] = 150
chutes[2] = 300

puts chutes[0] # imprime 50
puts chutes[1] # imprime 150
puts chutes[2] # imprime 300
```

Mas como sabemos o total de elementos que temos dentro de nosso array? Podemos usar um contador:

```
chutes = []
chutes[0] = 50
tentativa = 1

chutes[1] = 150
tentativa = 2

chutes[2] = 300
tentativa = 3

puts chutes[2] # imprime 300
puts chutes[tentativa - 1] # imprime 300
```

Ou ainda fazer um laço para imprimir todos os valores:

```
chutes = []
chutes[0] = 50
chutes[1] = 150
chutes[2] = 300
tentativa = 3

for contador in 1..tentativa
    puts chutes[tentativa - 1]
end
```

O resultado desse código é a impressão:

```
50
150
300
```

Mas se quisermos imprimir todos os valores de uma vez só, podemos pedir para imprimir diretamente o array:

```
chutes = []
chutes[0] = 50
chutes[1] = 150
```

```
chutes[2] = 300

puts chutes
```

Resultando em:

```
50
150
300
```

6.2 Aplicando o array ao nosso jogo

Vamos aplicar esse nosso array ao código do jogo. Primeiro criamos um array de chutes vazios, além do total de chutes:

```
limite_de_tentativas = 5
chutes = []
```

Assim que o usuário faz um chute, colocamos um valor dentro de nosso *array* conversando com ele: "Senhor `chutes`, por favor, coloque aí dentro o `chute` atual". Mas como? Da mesma maneira aumentamos o `total_de_chutes`:

```
for tentativa in 1..limite_de_tentativas
    chute = pede_um_numero tentativa, limite_de_tentativas

    chutes[total_de_chutes] = chute
    total_de_chutes = total_de_chutes + 1

    if verifica_se_acertou numero_secreto, chute
        break
    end
end
```

Como é tão comum somar em uma variável (além de outras operações matemáticas que veremos adiante), é possível somar e reatribuir o valor de uma vez só:

```
chutes[total_de_chutes] = chute
total_de_chutes += 1
```

Pronto, nosso *array* já está crescendo. Agora é hora de imprimi-lo a cada nova tentativa. Onde imprimimos todas as informações da rodada atual? No `pede_um_numero`, portanto passamos como parâmetro para ele o conteúdo de `chutes`:

```
for tentativa in 1..limite_de_tentativas
    chute = pede_um_numero chutes, tentativa,
            limite_de_tentativas
    chutes[total_de_chutes] = chute
    total_de_chutes += 1
    if verifica_se_acertou numero_secreto, chute
        break
    end
end
```

Em sua definição, recebemos os chutes como argumento, imprimindo-os:

```
def pede_um_numero(chutes, tentativa, limite_de_tentativas)
    puts "\n\n\n\n"
    puts "Tentativa " + tentativa.to_s + " de " +
          limite_de_tentativas.to_s
    puts "Chutes até agora: " + chutes
    puts "Entre com o número"
    chute = gets
    puts "Será que acertou? Você chutou " + chute
    chute.to_i
end
```

Ao rodar o código percebemos que, como de costume, *Array* não é convertido para *String* automaticamente através do operador `+`. Temos que invocar o método `to_s` para transformá-lo em uma *String*:

```
puts "Chutes até agora: " + chutes.to_s
```

Agora sim. Nosso código roda e funciona, imprimindo os números que chutamos até agora:

```
Tentativa 1 de 5
Chutes até agora: []
Entre com o número
```

```
200
...
Tentativa 2 de 5
Chutes até agora: [200]
Entre com o número
```

6.3 Facilidades de um array

Dado um dos exemplos de inserção de elementos em um array:

```
total = 0
chutes = []
chutes[total] = 50
total += 1
chutes[total] = 150
total += 1
chutes[total] = 300
total += 1

puts chutes
```

Que resulta em:

```
50
150
300
```

Será que um array não seria capaz de saber quantos elementos há dentro dele? Na prática, ele sabe, e para isso ele disponibiliza seu tamanho, seu *size*:

```
total = 0
chutes = []
chutes[total] = 50
total += 1
chutes[total] = 150
total += 1
chutes[total] = 300
total += 1

puts chutes.size # imprime 3
```

Ou ainda:

```
chutes = [100, 300, 500]
puts chutes.size # imprime 3

chutes[3] = 600
puts chutes.size # imprime 4
```

Se o tempo todo ele sabe o tamanho, seria possível incluir direto na primeira posição vazia. Note que a primeira posição vazia é a posição do total de elementos. Se tem zero elementos, é a posição zero. Se tem um elemento, é a posição um do array e por assim vai:

```
chutes = []
chutes[chutes.size] = 50
chutes[chutes.size] = 150
chutes[chutes.size] = 300

puts chutes.size # imprime 3
```

Mas se o array sabe seu total, não seria possível ele disponibilizar algo que permitisse incluir um elemento ao seu final, diretamente? Para isso temos o <<:

```
chutes = []
chutes << 50
chutes << 150
chutes << 300

puts chutes.size # imprime 3
```

Por fim, se quero fazer um laço por todos os elementos de um array, sou mesmo obrigado a criar uma variável temporária? Um contador? Quando contamos nos dedos usamos um contador. Mas se temos uma sacola com dez canetas, e queremos dizer a cor de cada uma delas, não usamos um contador, simplesmente abrimos o saco, pegamos uma caneta, olhamos a cor, pegamos a próxima caneta, olhamos a cor, a próxima, a próxima. Não precisamos contar "1", "2" etc. Quando queremos passar por todos os elementos, sem a necessidade de contar, podemos utilizar um `for` específico:

```
chutes = [100, 300, 500]
for chute in chutes
    puts chute
end
```

6.4 Simplificando nosso código de array

Essa sacada de manter uma variável de ajuda com o tamanho real utilizado de um array é muito importante e extremamente utilizado em diversas situações. Podemos remover nossa variável `total_de_chutes` e usar o `size`:

```
limite_de_tentativas = 5
chutes = []

for tentativa in 1..limite_de_tentativas
    chute = pede_um_numero chutes, tentativa,
            limite_de_tentativas
    chutes[chutes.size] = chute
    if verifica_se_acertou numero_secreto, chute
        break
    end
end
```

Mas se o Ruby tem algo para nos auxiliar com o tamanho atual de um array, será que ele já não tem algo que nos ajuda a colocar um valor no final dele? Sim, já vimos o símbolo `<<`:

```
chutes = [100, 300, 500]

chutes << 600

puts chutes.size # imprime 4
puts chutes[3] # imprime 600
```

Portanto, nosso código final fica ainda mais simples:

```
da_boas_vindas
numero_secreto = sorteia_numero_secreto
```

```
limite_de_tentativas = 5
chutes = []

for tentativa in 1..limite_de_tentativas
    chute = pede_um_numero chutes, tentativa,
            limite_de_tentativas
    chutes << chute
    if verifica_se_acertou numero_secreto, chute
        break
    end
end
```

Apesar de ser possível ensinar somente o `<<`, é importante entendermos que um array possui um tamanho inicial e que vamos adicionando aos poucos novos valores, mantendo um contador que diz quantos elementos temos lá dentro.

6.5 Arrays e Strings

Até agora usamos em diversos momentos algo que se comporta muito parecido a um array sem parar para pensar neles: eram nossas `String`, um conjunto de caracteres. `Strings` também possuem uma maneira de saber seu tamanho:

```
nome = gets
puts nome.size.to_s + " caracteres"
```

Rodamos o código com o nome *Guilherme* e ele imprime *10*. Dez? Mas meu nome tem nove caracteres! Testamos com diversos outros valores e o resultado parece ser sempre o mesmo, um caracter a mais. O que está acontecendo?

Vamos tentar imprimir o nome e ver o que existe lá dentro?

```
nome = gets
puts nome + " tem " + nome.size.to_s + " caracteres"
```

E o resultado:

```
Guilherme
Guilherme
 tem 10 caracteres
```

Que estranho! E se tentarmos imprimir o caractere que está nesta posição, utilizando o `[]` para pegar o elemento na posição `10`:

```
nome = gets
puts "Resultado: "
puts nome[10]
```

Resulta:

```
Guilherme
Resultado:
```

Realmente tem algo de estranho aqui. O que acontece? Vamos verificar o que eu digitei:

```
G, u, i, l, h, e, r, m, e, RETURN/ENTER
```

Acontece que quando lemos uma `String` com o `gets`, recebemos uma `String` incluindo o *return* (*enter*) que o usuário deu, isto é, a quebra de linha, o caractere de *new line* já está aí dentro. Por isso, toda `String` que lemos e que tentamos imprimir tem tamanho um pouco maior do que aparentemente digitamos. Na verdade nós também fornecemos como entrada o *return* e esse código também foi inserido em nossa `String`.

Como arrancar fora esse caractere? Existe uma função no Ruby que podemos usar com toda `String`. Essa função especial arranca os caracteres considerados "branco", ou seja, quebras de linha e espaço, tanto do começo quanto do final da sua `String`. Justamente por arrancar, o seu nome é `strip`. O código a seguir, que usa o `strip`, imprime `9`:

```
nome = gets.strip
puts nome + " tem " + nome.size.to_s + " caracteres"
```

Aplicamos o `strip` em nosso código ao ler o nome:

```
nome = gets.strip
```

E ao ler o chute:

```
chute = gets.strip
```

6.6 Interpolação de Strings

Outra característica interessante de uma `String` é a capacidade de criar uma `String` complexa com partes de outras variáveis. Tome como exemplo a impressão do nome que usamos agora:

```
nome = gets.strip
puts nome + " tem " + nome.size.to_s + " caracteres"
```

Começa a ficar bem chato ter que transformar tudo com `to_s` e ter que ficar concatenando com abre e fecha aspas, além de diversos `+`. Que confusão. Em vez disso, podemos criar uma nova `String` e colocar valores lá dentro:

```
nome = gets.strip
puts "#{nome} tem #{nome.size} caracteres"
```

O resultado final é o mesmo, mas o código fica bem mais simples. Essa característica de compor uma `String` nova com o resultado de outras variáveis é o que chamamos de interpolar uma `String` (*string interpolation*).

Podemos aplicar o conceito de interpolação de `String` em duas funcões de nosso jogo, simplificando nosso código ainda mais. A função `da_boas_vindas` pode fazer:

```
puts "Começaremos o jogo para você, #{nome}"
```

Já a função `pede_um_numero` fica:

```
def pede_um_numero(chutes, tentativa, limite_de_tentativas)
    puts "\n\n\n\n"
    puts "Tentativa #{tentativa} de #{limite_de_tentativas}"
    puts "Chutes até agora: #{chutes}"
    puts "Entre com o número"
    chute = gets.strip
    puts "Será que acertou? Você chutou #{chute}"
    chute.to_i
end
```

Pronto. Nosso código de concatenação passa agora a converter automaticamente, invocando a função `to_s` de cada um dos elementos que está sendo interpolado.

6.7 Funções e métodos

Até agora criamos funções soltas, funções que são definidas "voando" e portanto podem ser invocadas de qualquer lugar. Opa, parece a descrição de global, Guilherme? Isso mesmo. Nossas funções estão com um escopo que qualquer um pode invocar de qualquer lugar, o que pode parecer meio exagerado.

Veremos em outros exemplos como melhorar esse aspecto de nosso programa utilizando conceitos básicos de orientação a objeto. Mas antes de criarmos nossas funções com escopo controlado, já tivemos contato com elas, mesmo sem perceber!

Note que, quando limpamos uma `String` de caracteres brancos, não fizemos o código a seguir:

```
strip "  Guilherme  \n"
```

Mas sim:

```
"  Guilherme  \n".strip
```

Em vez de invocarmos uma função global, passando uma `String` como parâmetro, falamos para nosso objeto `String` que gostaríamos de chamar sua função `strip`. Fizemos isso através do operador `.`. Essa função `strip` não é global, mas específica de `String`s, portanto ela tem um escopo limitado e deve ser *sempre* aplicada a uma `String`. É como se a função, ao invés de ser uma coisa global, flutuante, sem dono e ao mesmo tempo com um grande dono (todo mundo, qualquer um), pertencesse a uma `String`, como se ela fosse um comportamento de uma `String`. Toda `String` tem a capacidade de ser *strip*ada, tem esse comportamento disponível, esse método (*method*) disponível.

6.8 Testando métodos no IRB

Mas quem pode ter um método? Qualquer valor tem métodos?

Vamos fazer alguns testes e para isso gostaria de apresentar uma ferramenta do Ruby que permite a execução de pequenos testes de código rapidamente, o *irb*. Executando *irb*, temos uma linha de comando onde podemos

executar qualquer comando Ruby, e ele mostra a saída e o retorno daquele código.

Um teste fácil é chamar o método *methods*, que devolve a lista de métodos que um valor em Ruby possui. Repare os métodos que minha `String` tem:

```
irb(main):001:0> "Guilherme".methods
=> [:<=>, :==, :===, :eql?, :hash, :casecmp, :+, :*, :%, :[],
:[]=, :insert, :length, :size, :bytesize, :empty?, :=~, :match,
:succ, :succ!, :next, :next!, :upto, :index, :rindex, :replace,
:clear, :chr, :getbyte, ...
:enum_for, :equal?, :!, :!=, :instance_eval, :instance_exec,
:__send__, :__id__]
```

E os métodos de um número inteiro:

```
irb(main):002:0> 175.method
=> [:to_s, :inspect, :-@, :+, :-, :*, :/, :div, :%, :modulo,
:divmod, :fdiv, :**, :abs, :magnitude, :==, :===, :<=>, :>, :>=,
:<, :<=, :~, :&, :|, :^, :[], :<<, :>>, :to_f, :size, :zero?,
:odd?, :even?, :succ, :integer?, :upto, ...
:enum_for, :equal?, :!, :!=, :instance_eval, :instance_exec,
:__send__, :__id__]
```

Muita coisa? Não tem problema, aos poucos vamos conhecendo e utilizando cada vez mais deles. Por exemplo, já usamos o método `strip` e `to_i` de uma `String`. Já usamos o `to_s` de um inteiro. Poderíamos também usar o `upcase` de uma `String` para receber uma outra `String`, agora em letra maiúscula:

```
nome = "Guilherme"
puts nome.upcase # GUILHERME
```

Ou o método `odd?` e `even?` para saber se um número é ímpar ou par:

```
numero_secreto = 175
puts "O números secreto é ímpar?"
puts numero_secreto.odd? # true
```

Cada valor em Ruby é um objeto, e todo objeto pode ter métodos, por isso basicamente todo valor em Ruby pode ter métodos. Veremos como criar

nossos próprios tipos e objetos mais adiante, por enquanto é suficiente entendermos que todos os nossos valores são de um tipo determinado e por isso possuem diversos métodos que podemos invocar.

Olhando a lista de métodos de um número inteiro, uma `String` e um `Array`, surgem símbolos que já utilizamos, como os `[]` para acessar uma posição de um array, o `+` para concatenar ou somar, ou ainda o `<<` e o `==`. Seriam eles métodos? Sim, na prática eles são métodos e poderíamos invocá-los como tal, só não parece nada natural:

```
irb(main):001:0> "Guilherme".+(" Silveira")
=> "Guilherme Silveira"

irb(main):002:0> x = [50, 100, 150]
=> [50, 100, 150]

irb(main):003:0> x.[](2)
=> 150

irb(main):004:0> x.<< 200
=> [50, 100, 150, 200]
```

Lembra que o símbolo `+` não funciona entre uma `String` e um inteiro? É justo pois o método `+` de uma `String` é para concatenar `Strings`. Já o método `+` de números inteiros é para somar inteiros. Se só existisse uma função global que se chamasse `+`, como fazer para ela suportar todos os tipos de `+` que existem no mundo?

O método serve para colocar comportamento no valor, juntar o comportamento específico ao valor específico: uma `String` sabe se tornar maiúscula, e um número sabe dizer se é par ou ímpar.

6.9 Resumindo

Vimos como armazenar diversos dados em uma única variável através do uso de um array. Por trás dos panos, esse array possui um número determinado de espaços e um contador, e à medida que ele é ocupado, o espaço dobra de tamanho permitindo colocar mais coisas lá dentro. Vimos que `Strings` e

números inteiros são valores que possuem métodos, funções que possuem um escopo específico, e justamente por isso temos funções que parecem ser iguais (como a função `+`) mas na verdade são diferentes, só que com o mesmo nome. Conhecemos diversas funções e métodos já, desde o `puts` e `gets` até o `strip` e o `odd?`.

Vimos também como lidar com um array na prática, adicionando e buscando elementos, além de pesquisando seu tamanho. Aprendemos a utilizar a interpolação de `String` para facilitar a criação de textos customizados. Por fim, revisamos por cima como um interpretador de Ruby pode gerenciar a memória de variáveis simples e arrays.

Capítulo 7

Pontos e matemática

Chegou a hora de mostrar para o nosso jogador quantos pontos ele ganhou. O jogador começa com mil pontos. A cada chute que ele faz ele perde uma parte desses pontos. Por exemplo, se o número é o `175` e ele chuta `215`, ele errou por `40`, então o jogador perde `20` pontos, isto é, ele perde a diferença entre os dois números dividida por dois.

Sabemos definir uma variável como um número, portanto os pontos até esse instante são `1000`:

```
limite_de_tentativas = 5
chutes = []
pontos_ate_agora = 1000
```

A cada novo chute, calculamos a diferença que erramos, `chute` menos o `numero_secreto`:

```
pontos_a_perder = chute - numero_secreto
```

Agora queremos dividir (`/`) este valor por `2`:

```
pontos_a_perder = chute - numero_secreto / 2
```

Por fim, tiramos dos pontos até agora os pontos a perder:

```
pontos_ate_agora = pontos_ate_agora - pontos_a_perder
```

No final do programa imprimimos quantos pontos ele ganhou:

```
puts "Você ganhou #{pontos_ate_agora} pontos."
```

Ficando com o seguinte código:

```
da_boas_vindas
numero_secreto = sorteia_numero_secreto

limite_de_tentativas = 5
chutes = []
pontos_ate_agora = 1000

for tentativa in 1..limite_de_tentativas
    chute = pede_um_numero chutes, tentativa,
            limite_de_tentativas
    chutes << chute

    pontos_a_perder = chute - numero_secreto / 2
    pontos_ate_agora = pontos_ate_agora - pontos_a_perder
    if verifica_se_acertou numero_secreto, chute
        break
    end
end

puts "Você ganhou #{pontos_ate_agora} pontos."
```

Sabendo que o número secreto é `175`, testaremos primeiro chutar o número `215`. Devemos perder `20` pontos. Agora vamos acertar, o que nos dará `980` pontos, repare a saída:

```
...
Entre com o número
215
...
Entre com o número
175
...
Acertou!
Você ganhou 784 pontos.
```

Como assim? Acontece que, assim como na matemática, as operações de soma (+), subtração (-), multiplicação (*), divisão (/) etc. obedecem uma ordem rígida. Antes a divisão e multiplicação, depois a soma e subtração. Portanto nosso código a seguir está dividindo o número secreto por 2:

```
pontos_a_perder = chute - numero_secreto / 2
```

O que faz com que a conta tenha um resultado totalmente diferente do que queríamos. Nós queremos que a divisão seja feita depois. Como era feito na escola mesmo? Com o uso de parênteses para indicar quem deve ser executado antes, para mudar a precedência padrão do interpretador:

```
pontos_a_perder = (chute - numero_secreto) / 2
```

Agora sim:

```
...
Entre com o número
215
...
Entre com o número
175
...
Acertou!
Você ganhou 980 pontos.
```

7.1 Ponto flutuante

Vamos testar com outros números agora. Que tal tentarmos `176` e depois `175`? Devemos perder somente meio ponto, resultando em `999.5`:

```
...
Entre com o número
176
...
Entre com o número
175
...
Acertou!
Você ganhou 1000 pontos.
```

Como assim? Não perdi nada? Esse resultado só é possível se `pontos_a_perder` for zero. Mas ele é a divisão de um por dois. Quanto é um dividido por dois? A resposta é: depende.

Na escola aprendemos isso e facilmente esquecemos quando crescemos. Primeiro aprendemos que a divisão de um por dois é zero e dá resto um. Depois aprendemos que a divisão de um por dois é meio. As duas respostas são válidas: uma no universo dos números inteiros, e a outra no universo dos números reais (no nosso caso, com *ponto flutuante*).

Por que o interpretador nos levou ao primeiro caso? Pois até agora usamos números inteiros. Por exemplo:

```
irb(main):001:0> 215 - 175
=> 40
irb(main):002:0> 215 - 175 / 2
=> 128
irb(main):003:0> (215 - 175) / 2
=> 20
irb(main):004:0> exit
```

Mas vamos para nosso caso atual, em que usamos `176`:

```
irb(main):001:0> 176 - 175
=> 1
irb(main):002:0> (176 - 175) / 2
=> 0
```

A conta está sendo feita com dois números inteiros, portanto o resultado é inteiro. Mas, Guilherme, você consegue me provar que esse número está

sendo tratado como um inteiro? Sim, o método `.class` indica qual o tipo daquele valor que estamos referenciando. Repare que números inteiros são `Fixnum` e números com ponto flutuante são `Float`:

```
irb(main):001:0> chute = 176
=> 176
irb(main):002:0> chute.class
=> Fixnum
irb(main):003:0> chute = 176.0
=> 176.0
irb(main):004:0> chute.class
=> Float
```

Se tenho pelo menos um float, o computador sabe que a conta deve ser feita com número flutuante:

```
irb(main):001:0> 1.0 / 2
=> 0.5
irb(main):002:0> 1 / 2.0
=> 0.5
irb(main):003:0> 1.0 / 2.0
=> 0.5
```

Se ambos são números inteiros, o computador usa números inteiros:

```
irb(main):001:0> 1 / 2
=> 0
```

Sabendo disso, como corrigir nosso código para a conta ser feita com números com ponto flutuante? Uma solução é dividir por `2.0`, a outra é multiplicar por `0.5`, ambas teriam o mesmo efeito prático:

```
pontos_a_perder = (chute - numero_secreto) * 0.5
pontos_a_perder = (chute - numero_secreto) / 2.0
```

Mantemos a opção da divisão por `2.0` por deixar mais claro o que estamos fazendo com a diferença do chute para o próximo desenvolvedor, lembra-se da importância de pequenas escolhas na compreensão do código? Agora sim o resultado é o número com ponto flutuante adequado:

```
...
Entre com o número
176
...
Entre com o número
175
...
Acertou!
Você ganhou 999.5 pontos.
```

7.2 Simulação do código

Vamos tentar agora outro cenário, chutemos `171` e `175`:

```
...
Entre com o número
171
...
Entre com o número
175
...
Acertou!
Você ganhou 1002.0 pontos.
```

Como é possível eu ter ganhado pontos? Vamos revisar a fórmula que escrevemos:

```
# ...
pontos_ate_agora = 1000 # linha 1

for tentativa in 1..limite_de_tentativas # linha 2
    chute = # ... linha 3

    pontos_a_perder = (chute - numero_secreto) / 2.0 # linha 4
    pontos_ate_agora = pontos_ate_agora -
                       pontos_a_perder # linha 5
    # ... linha 6
end
# linha 7
```

Quero saber exatamente o que acontece para chegar no número `1002.0`, isto é, quero simular meu programa na mão. Para isso, pegamos um pedaço de papel (ou um pedaço da nossa memória, na nossa cabeça), e começamos a "executar" o código no papel, linha a linha.

O programa chega à linha marcada `1`. Até aqui, não imaginamos que o código executado vai interferir com nosso resultado. Agora executamos a linha `1`, chegando à linha `2`. O que temos na nossa memória? A seguinte situação, em que existem variáveis com valores:

```
limite_de_tentativas = 5
pontos_ate_agora = 1000
```

Executamos a linha `2`, entrando na linha `3`:

```
limite_de_tentativas = 5
pontos_ate_agora = 1000
tentativa = 1
```

Executamos a linha `3`, entrando com o número `171`:

```
limite_de_tentativas = 5
pontos_ate_agora = 1000
tentativa = 1
chute = 171
```

Portanto, a linha `4` será equivalente a:

```
pontos_a_perder = (171 - 175) / 2
pontos_a_perder = -2
```

Opa. Número negativo? Vamos executá-la:

```
limite_de_tentativas = 5
pontos_ate_agora = 1000
tentativa = 1
chute = 171
pontos_a_perder = -2
```

Finalmente, executamos a linha `5`:

```
pontos_ate_agora = 1000 - -2
pontos_ate_agora = 1002
```

Encontramos nosso bug. O número negativo ocorreu pois o número chutado foi menor que o número real. Esse processo de simular no papel o código sendo executado, junto com os dados das variáveis na memória, é vital no começo de carreira de um desenvolvedor para ser capaz de mais para a frente inferir o comportamento do código sem precisar do papel, somente de cabeça. O exemplo dado aqui é propositalmente mais simples do que os exemplos futuros que veremos, mas mesmo no dia a dia erros de conta negativa, divisão por zero, soma mais ou menos um, são extremamente comuns. Um desenvolvedor com forte treino de simulação é capaz de detectar e resolver tais problemas mais facilmente. Não tenha vergonha.

7.3 Matemática

Vamos corrigir nosso programa verificando se a diferença é negativa:

```
pontos_a_perder = (chute - numero_secreto) / 2.0
if pontos_a_perder < 0
    pontos_a_perder = -pontos_a_perder
end
pontos_ate_agora = pontos_ate_agora - pontos_a_perder
```

Já conhecemos o outro estilo de `if`, o de uma única linha, que poderia ser utilizado aqui:

```
pontos_a_perder = (chute - numero_secreto) / 2.0
pontos_a_perder = -pontos_a_perder if pontos_a_perder < 0
pontos_ate_agora = pontos_ate_agora - pontos_a_perder
```

Mas evitaremos esse código, ele é muito obscuro pois esconde o `if` de nossa vista. Mesmo não usando uma variação do `if` de uma linha, podemos fazer uma variação com a conta de vezes que acredito pessoalmente ficar mais clara. Se o número é negativo, inverte o sinal multiplicando o próprio número por `-1`:

```
pontos_a_perder = (chute - numero_secreto) / 2.0
if pontos_a_perder < 0
    pontos_a_perder *= -1
end
pontos_ate_agora = pontos_ate_agora - pontos_a_perder
```

Ainda não está bom? Voltemos à aula de matemática. Quando temos um número e queremos saber seu valor, removido o sinal, queremos seu valor *abs*oluto. *Abs*oluto, entendeu? Dentre os métodos de números inteiros, existe uma função, o método *abs*:

```
pontos_a_perder = (chute - numero_secreto).abs / 2.0
pontos_ate_agora = pontos_ate_agora - pontos_a_perder
if verifica_se_acertou numero_secreto, chute
    break
end
```

Pronto. A diferença, independente de número, dividida por `2`.

7.4 Unless... e a dupla negação

Outra variação seria utilizar o inverso de um `if`, o somente se não, o `unless`. Somente se não for maior que zero:

```
pontos_a_perder = (chute - numero_secreto) / 2.0
unless pontos_a_perder > 0
    pontos_a_perder = -pontos_a_perder
end
pontos_ate_agora = pontos_ate_agora - pontos_a_perder
```

Ou ainda em uma linha:

```
pontos_a_perder = (chute - numero_secreto) / 2.0
pontos_a_perder = -pontos_a_perder unless pontos_a_perder > 0
pontos_ate_agora = pontos_ate_agora - pontos_a_perder
```

Não se incomode se o `unless` não parecer natural. Ele não é. Nada negativo é natural. Repare as duas frases a seguir:

```
O natural é entendermos uma frase afirmativa.
O natural é entendermos uma frase não negativa.
```

A primeira é mais simples do que a segunda por um motivo: ela não envolve uma negação. Uma das piores práticas é ainda o uso da dupla negação, mais confusa ainda:

```
O natural não é entendermos uma frase não afirmativa.
```

Pior ainda se a negação é implícita. O exemplo a seguir continua sendo o mesmo de sempre, mas a dupla negação parece escondida:

```
O natural é não entendermos uma frase negativa.
```

Evitaremos o uso do `unless` por envolver uma negação implícita (como no uso da palavra *negativa*) ao invés de explícita (como no uso do *não*). E mesmo que fosse explícito, evitaremos o uso da dupla negação. No fim, use o `unless` somente se ele for mudar o sentido de sua vida espiritual: raramente. Portanto não alteramos em nada nosso código.

7.5 Número aleatório

Até agora o nosso jogo utilizou o número `175` de maneira fixa, mas já está na hora de escolhermos um número qualquer, um número aleatório, que toda vez muda. Existe uma função em Ruby chamada `rand` que devolve um número entre `0` (inclusive) e `1` (exclusive). Portanto, poderíamos calcular:

```
aleatorio = rand
```

Para que nosso número tenha o valor mínimo `0` (inclusive) e máximo `200` (exclusive) podemos multiplicar o número por `200`:

```
aleatorio = rand
numero_secreto = aleatorio * 200
```

Essa abordagem é bastante utilizada em diversas linguagens, mas em Ruby temos a opção de passar como argumento para a função `rand` o número `200`, obtendo o mesmo tipo de resultado que antes:

```
numero_secreto = rand(200)
```

Nossa função `sorteia_numero_secreto` fica:

```
def sorteia_numero_secreto
    puts "Escolhendo um número secreto entre 0 e 200..."
    sorteado = rand(200)
    puts "Escolhido... que tal adivinhar hoje nosso número
          secreto?"
    sorteado
end
```

7.6 Operadores matemáticos

Vimos até aqui diversos operadores matemáticos que podem ser aplicados tanto a valores do tipo `Fixnum` quanto `Floats`. As quatro operações básicas são a soma (+), subtração (-), multiplicação (*) e divisão (/). Uma quinta operação importante é a de resto (módulo, *mod*,

```
treze = 13
cinco = 5

dois = treze / cinco
tres = treze % cinco
dois_ponto_seis = 13.0 / 5
```

Esses operadores costumam ser os mesmos em quase todas as linguagens de programação. Além deles, utilizamos o parênteses para redefinir a prioridade de execução das operações.

Existem atalhos para operações tradicionais, como o caso de somar, subtrair, dividir, multiplicar ou aplicar o resto de uma divisão em si mesmo:

```
numero = 5
numero += 1 # 6
numero -= 2 # 4
numero *= 2 # 8
numero /= 4 # 2
numero *= 13 # 26
numero %= 10 # 6
```

7.7 Sistema de tipos

Como vimos até agora, cada variável que criamos referencia um valor de um tipo determinado. Cada linguagem trabalha com um sistema de tipos diferentes. Em Ruby, uma variável pode referenciar em um instante uma variável de um tipo `Fixnum` e depois uma de um tipo `String`:

```
quinze = 15
quinze = "Quinze"
```

Porém, o que importa em tempo de execução é o tipo do valor referenciado:

```
quinze = "  Quinze  "
quinze.strip # Quinze
quinze = 15
quinze.strip # undefined method `strip'
```

Cada valor é uma instância daquele tipo, é um objeto daquele tipo, portanto pode responder a diversos métodos que foram definidos nele. Veremos no futuro como criar nossos próprios tipos e métodos.

Em Ruby, um tipo é dinâmico o suficiente para que ele possa ser alterado, e um método que antes não existia pode passar a existir. Esse conceito, de que uma categoria de valores é aberta para modificação, (chamado de *classe aberta*), ou seja, essa capacidade que permite a alteração de um tipo em tempo de execução é poderosa e perigosa, pois pode surtir efeitos inesperados, como quebra de compatibilidade, difíceis de serem simulados e compreendidos.

Você pode consultar a documentação de cada tipo, como da classe `String` no site da linguagem: http://ruby-doc.org/core-2.1.2/String.html

7.8 Resumindo

Aprendemos a fazer operações matemáticas com números inteiros e com pontos flutuantes (casas decimais em geral), além de colocarmos em prática um dos processos que fará parte do seu dia a dia de programação: como simular um programa primeiro no papel, depois, com a prática, de cabeça. A simulação ocorre cada vez mais de maneira automática mas um código com

alta complexidade torna tal tarefa difícil, por isso mesmo a importância de sempre refatorarmos e tentarmos simplificar o código final.

Aprendemos a utilizar o `unless` e a tomar cuidado para evitarmos uma dupla negação, além de como utilizar números aleatórios e entender que, se eu tenho tipos diferentes, como `String`, `Array` etc., eu preciso de um sistema de tipos, uma arquitetura de como devem funcionar e se comportar esses tipos, além de suas relações.

Capítulo 8

Binário

8.1 Binário

Já vimos que o computador usa a memória para armazenar os valores e arrays que utilizamos em nosso programa. Mas sempre ouvimos falar que ele usa somente zero e uns para fazer tudo, como assim?

A ideia de que zeros e uns em quantidade suficiente podem representar qualquer coisa é inicialmente assustadora. Imagine o número 0, como representá-lo? Fácil: 0. E o número 1? Também é fácil, 1. Pronto, temos um *bit*.

O desafio começa, claro, quando queremos representar o número 2, usando somente os algarismos 0 e 1. A solução é usar o 10 para representar o número 2. Cuidado. Não leia `10` como dez, leia como "um zero". E o número três? 11. Isto é, "um um". E depois o número quatro? 100. Depois 101, 110, 111. Ficamos com uma tabela:

```
0 =>    0
1 =>    1
2 =>   10
3 =>   11
4 =>  100
5 =>  101
6 =>  110
7 =>  111
8 => 1000
...
```

Claro, assim como na matemática tradicional, podemos preencher com zero à esquerda para padronizar a quantidade de dígitos. Por exemplo, se usarmos 8 dígitos, 1 *byte*:

```
0 => 00000000
 1 => 00000001
 2 => 00000010
 3 => 00000011
 4 => 00000100
 5 => 00000101
 6 => 00000110
 7 => 00000111
 8 => 00001000
 9 => 00001001
10 => 00001010
11 => 00001011
12 => 00001100
13 => 00001101
14 => 00001110
15 => 00001111
```

Pronto. Podemos seguir o mesmo padrão até qualquer número inteiro, claro, dado que tenhamos memória suficiente para armazenar esse número como uma sequência de zeros e uns. Não precisamos decorar agora a fórmula de conversão da base 01, da base de dois algarismos (*binária*), para a base 0123456789, a base de dez algarismos (*decimal*). Nem a de ida, nem a de volta. Mas já somos capazes de entender como o computador representa todos os números inteiros.

8.2 Binário e letras

Mas como representar letras? E números com ponto flutuante, casas decimais?

Dê um número para uma letra. Por exemplo, 'A = 65'. Então 'B = 66', 'C = 67':

```
A => 65 => 01000001
B => 66 => 01000010
C => 67 => 01000011
...
Z => 90 => 01011010
```

Pronto, o computador é capaz de representar todas as letras existentes no nosso alfabeto usando somente 8 dígitos 0 ou 1. Essa tabela do nosso alfabeto, incluindo diversos outros caracteres, é a tabela ASCII, usada por muito tempo como o principal padrão de tradução de números e caracteres por um computador.

8.3 Bits: 8, 16, 32, 64

Cada um desses números é um bit, uma unidade 0 ou 1. Se estamos utilizando oito bits para representar um número inteiro, podemos representar 2*2*2*2*2*2*2*2 = 256. Somente os números zero a 256 podem ser representados com oito bits. Que horror. Mas era assim que funcionava o Master System. Já o Mega Drive tinha um processador de 16 bits, representava nativamente números até 65.536. Não é à toa que os jogos do Master tinham em geral no máximo 256 cores, enquanto as do Mega tinham até 65 mil.

Mas mesmo esse número, será que ele é capaz de representar todos os números inteiros que queremos? Ou ainda um processador capaz de entender e armazenar 16 bits é capaz de representar todas as letras de todos os alfabetos do mundo?

Os processadores modernos são capazes de representar números com até 64 bits, o que dá bons bilhões, além de todo o conjunto de caracteres dos alfabetos existentes no mundo. Mesmo assim existem otimizações para representar o alfabeto, como um padrão famoso por ter substituído o ASCII em 2008 como o mais utilizado na internet, o UTF-8.

8.4 Bits e números com ponto flutuante

Mas a questão do ponto flutuante, da casa decimal, ainda persiste. Como representar tais números se o computador só conhece zeros e uns, números inteiros? É necessário inventar algum padrão. Vamos criar o nosso próprio, mais simples do que um usado hoje em dia. Imagine o número treze e meio, também conhecido como 13,5. Quantos dígitos ele possui antes da casa decimal? Dois? Então, `0010`. Qual o valor que vem antes da casa decimal? Um depois três? Então, `0001 0011`. Qual o dígito que vem depois da casa decimal? `5`? Então, `0101`. Portanto, para representar o número `13,5` utilizarei a notação que indica o número de casas antes da vírgula (`2`), o valor delas, dígito a dígito (`1` e `3`) e o valor decimal (`5`):

```
0010 0001 0011 0101
```

Pronto. O computador é capaz de entender esse número como `13,5`. Com isso somos capazes de representar qualquer número com casa decimal cuja quantidade de dígitos seja finita, mesmo que essa técnica esteja longe de ser ótima ou perfeita. Existe uma limitação grande em relação à representação de números com ponto flutuante. Se eu uso esse tipo de abordagem, como representar o valor `1 / 3`, isto é, um terço?

Esse número tem uma quantidade infinita de dígitos. Não podemos representar na nossa abordagem. O que fazer? Arredondamos para `0,3`, perdemos precisão:

```
0001 0000 0011
```

O arredondamento foi muito ruim? Tentamos então `0,33`:

```
0001 0000 0011 0011
```

Ou ainda `0,333`:

```
0001 0000 0011 0011 0011
```

Como o computador costuma seguir padrões de tamanho fixo, é comum hoje em dia que um número de ponto flutuante, assim como um número inteiro, utilize 64 bits. Isso limita a nossa aproximação. Para nosso um terço, usando essa ideia apresentada (não ideal), temos a melhor aproximação de `0,333.3` com `62` casas após a vírgula.

```
0001 0000 0011 0011 0011 ... 0011
```

Em nossa abordagem de representação (em nosso algoritmo) isso nos limita bastante, mas a representação escolhida por um processador é bem mais inteligente que a nossa e permite aproximações muito boas.

Uma ideia que surge é a de representar uma divisão como essa com `1` dividido por `3`: `0001 0011`, e a ideia é boa. Existem sistemas e linguagens, principalmente matemáticas, que vão valorizar a precisão mais do que a memória ou a performance. As linguagens em geral não fazem isso por padrão.

8.5 Hexadecimal

Assim como vimos a base binária, que tem somente dois algarismos distintos para representar *todos* os números, conhecemos a decimal, que tem dez algarismos. Existe outra base bastante utilizada, a base `hexadecimal`, que tem dezesseis algarismos. Isso pois ela é o conjunto de quatro bits, que representa dezesseis possibilidades diferentes, como vimos antes.

Mas se temos dezesseis algarismos, que algarismos são esses? Na escola eu só vi do 0 até o 9. Aqui, usaremos as letras A, B, C, D, E e F para representar esses outros seis algarismos. Novamente, sem a necessidade de decorar a fórmula de transformação, a tabela a seguir mostra os `16` primeiros números em base hexadecimal:

```
0 => 0
 1 => 1
 2 => 2
 3 => 3
 4 => 4
 5 => 5
 6 => 6
 7 => 7
 8 => 8
 9 => 9
10 => A
11 => B
12 => C
13 => D
```

```
14 => E
15 => F
```

E poderíamos continuar, com:

```
16  => 10
17  => 11
18  => 12
...
32  => 21
...
177 => B1
...
255 => FF
```

Portanto, com dois algarismos de base hexadecimal (16 possibilidades cada), podemos representar `16 * 16` números, `256` números.

8.6 Bits e imagens

E o último desafio: como representar uma imagem de uma maneira bem simples? Se somos capazes de mapear uma coisa a números inteiros, somos capazes de mapear ao mundo binário, certo?

Nosso desafio é como representar uma imagem com números inteiros. Primeiro vamos pensar em uma foto. Ela tem diversos pontos. Cada ponto tem uma cor diferente. Imagine que a foto tem 1000 por 1000 pontos. O que podemos fazer é primeiro enumerar todas as cores existentes, limitando-as a 256:

```
cor0   = verde
cor1   = vermelho
cor2   = azul
cor3   = preto
cor4   = branco
...
cor255 = cinza
```

Mas calma, Guilherme, você disse que o computador entende números, não cores. Sim, e cada cor no computador tem um número. Por exemplo:

```
cor0   = 00FF00
cor1   = FF0000
cor2   = 0000FF
cor3   = 000000
cor4   = FFFFFF
...
cor255 = CCCCCC
```

Pronto. Dada essa tabela, agora podemos escrever uma linha para cada ponto na foto. Se o primeiro ponto da foto é vermelho (cor1) e o segundo azul (cor2) e o terceiro cinza (cor255) etc., em uma imagem com diversas linhas temos:

```
cor1 cor2 cor255 cor3 cor3 cor2 cor3 cor255
cor1 cor3 cor255 cor3 cor2 cor3 cor3 cor255
...
```

Que em hexadecimal seria:

```
01 02 FF 03 03 02 03 FF
01 03 FF 03 02 03 03 FF
...
```

A sequência `0102FF` representa uma pequena imagem de 3 pontos, cujas cores seriam vermelho, azul e cinza. Já `0102FF03030203FF\n0103FF03020303FF` representa uma imagem com duas linhas e 8 colunas.

Com isso podemos representar qualquer imagem com tamanho finito (que caiba na memória) e cujo número de cores esteja limitado a `256`. Esse é o raciocínio atrás de um mapa de bits de cores, um *bitmap*, o formato `BMP`, tanto utilizado antigamente no mundo da computação. Sua adoção não é mais tão grande devido ao tamanho que ele ocupa: uma imagem com grande qualidade exige uma tabela de cores enorme, e um número de pontos maior ainda. Algoritmos de compressão como o `JPEG` dominam esse mercado.

Assim como pensamos na alternativa de representar um terço como uma fração (um dividido por três), poderíamos representar uma imagem como uma sequência de traços, círculos etc. Essa também é uma ideia boa para

diversos tipos de imagem, e é o raciocínio, o algoritmo, por trás das imagens vetoriais, como o *SVG*.

O mesmo pode ser aplicado ao mundo da música, com o `WAV` e `MP3`, aos filmes etc.

No fim, representamos desde um número inteiro até uma imagem de tomografia com zeros ou uns. Temos uma função, um algoritmo, que mapeia um mundo ao outro.

8.7 Resumindo

Vimos como um computador é capaz de representar números inteiros em um formato binário, com somente dois algarismos, e escolhemos 0 e 1 para serem esses algarismos. Aprendemos que nossa base do dia a dia é a decimal, com os algarismos de 0 a 9. Vimos que ele utiliza a base hexadecimal com os algarismos extras A até F, e que convertendo de um para o outro passamos a ser capazes de converter qualquer número em algo para o computador, sempre limitado a determinado tipo de arredondamento, uma vez que temos limite de memória.

Aprendemos como funciona até mesmo uma função genérica que transforma caracteres em números e ainda imagens em números, tudo isso para entender o poder da matemática na computação: e isso é só o início para aqueles que pretendem se aventurar no mundo da pesquisa na ciência da computação.

Capítulo 9

Nível de dificuldade e o case

Nosso jogo ainda é bem difícil. Um número entre zero e duzentos precisa em média mais de 5 tentativas para acertar. Isso pois uma tática comum (e inteligente) é começar com o número do meio: 100.

Se errar e o número for maior, tente o novo número do meio, entre 100 e 200: 150. Novamente, 175. Novamente, 187. Para o último chute, teríamos as opções entre 188 e 200, uma chance muito pequena de acertar.

Vamos fazer como todos os jogos fazem: permitir ao jogador que escolha o nível de dificuldade, cada um determinando o intervalo de números dentro do qual será sorteado o nosso número secreto. Por exemplo, no nível fácil, escolheremos um número entre 0 e 30. No mais difícil, um número entre 0 e 200.

Para definir nossa função `pede_dificuldade`:

```
def pede_dificuldade
```

```
    puts "Qual o nível de dificuldade que deseja?
          (1 fácil, 5 difícil)"
    dificuldade = gets.to_i
end
```

Sabemos que o Ruby sempre retorna o resultado da última expressão, portanto não precisamos da definição da variável `dificuldade`:

```
def pede_dificuldade
    puts "Qual o nível de dificuldade que deseja?
          (1 fácil, 5 difícil)"
    gets.to_i
end
```

Porém esse código é mais difícil de compreender e nos salva somente menos de 15 caracteres. Não machuca deixá-lo mais claro, por isso fugirei aqui do padrão de não definir a variável, deixando óbvio o retorno da função:

```
def pede_dificuldade
    puts "Qual o nível de dificuldade que deseja?
          (1 fácil, 5 difícil)"
    dificuldade = gets.to_i
end
```

Agora quando começamos o jogo, logo depois de dar boas-vindas, pediremos o nível de dificuldade:

```
da_boas_vindas
dificuldade = pede_dificuldade
numero_secreto = sorteia_numero_secreto
```

Para o sorteio do número secreto será necessário saber a dificuldade, portanto a passamos como parâmetro:

```
da_boas_vindas
dificuldade = pede_dificuldade
numero_secreto = sorteia_numero_secreto dificuldade
```

Na nossa função `sorteia_numero_secreto` o recebemos:

```
def sorteia_numero_secreto(dificuldade)
    puts "Escolhendo um número secreto entre 0 e 200..."
    sorteado = rand(200)
    puts "Escolhido... que tal adivinhar hoje nosso número
          secreto?"
    sorteado
end
```

Definimos agora nossa variável `maximo` e corrigimos um bug. Nosso número máximo não era `200`, ele era `199`:

```
def sorteia_numero_secreto(dificuldade)
    maximo = ???
    puts "Escolhendo um número secreto entre 0 e
          #{maximo - 1}..."
    sorteado = rand(maximo)
    puts "Escolhido... que tal adivinhar hoje nosso número
          secreto?"
    sorteado
end
```

Precisamos agora verificar o valor da dificuldade e configurar o máximo de acordo com ela:

```
if dificuldade == 1
    maximo = 30
else
    if dificuldade == 2
        maximo = 60
    else
        if dificuldade == 3
            maximo = 100
        else
            if dificuldade == 4
                maximo = 150
            else
                maximo = 200
            end
        end
```

```
    end
end
```

Sim. Que nojo. Seria possível fazer uma série de `ifs`? Algo que, caso o primeiro fosse falso, tentasse o segundo, caso o segundo fosse falso, tentasse o terceiro? Sim, a maior parte das linguagens suporta a construção que junta o `else` com um novo `if`. Nesse caso, a sequência `if...elsif...end` executará somente a primeira condição verdadeira:

```
if dificuldade == 1
    maximo = 30
elsif dificuldade == 2
    maximo = 60
elsif dificuldade == 3
    maximo = 100
elsif dificuldade == 4
    maximo = 150
else
    maximo = 200
end
```

9.1 Case...when...end

Por mais que o `if...elsif...end` resolva nosso problema, como repetimos a restrição sempre de acordo com a mesma variável (`dificuldade`), existe outra construção, ainda mais simples. Caso a dificuldade seja 1, o máximo é 30. Caso seja 2, 60 etc.

```
case dificuldade
when 1
    maximo = 30
when 2
    maximo = 60
when 3
    maximo = 100
when 4
    maximo = 150
else
```

```
    maximo = 200
end
```

Eu também poderia juntar dois de uma vez só, mas não vou, pois o código fica ruim novamente:

```
case dificuldade
when 1..2
    maximo = 30 * dificuldade
when 3
    maximo = 100
when 4
    maximo = 150
else
    maximo = 200
end
```

Rodamos nosso jogo e verificamos que ele funciona. Ele mostra o máximo de acordo com o nível de dificuldade e escolhe um número naquele intervalo.

9.2 Escopo de variável local

Mas repare algo de estranho. A variável `maximo` foi definida dentro do `when`, e foi utilizada fora dele:

```
def sorteia_numero_secreto(dificuldade)
    case dificuldade
    when 1
        maximo = 30
    when 2
        maximo = 60
    when 3
        maximo = 100
    when 4
        maximo = 150
    else
        maximo = 200
    end
    puts "Escolhendo um número secreto entre 1 e #{maximo}..."
```

```
    sorteado = rand(maximo) + 1
    puts "Escolhido... que tal adivinhar hoje nosso número
          secreto?"
    sorteado
end
```

Em Ruby, uma variável definida dentro de uma função é visível durante toda a função. Não é um `if`, `case` ou `for` que limita o escopo da variável mais ainda. Por exemplo, o código a seguir imprime 15:

```
for i in 1..15
    puts "calculando"
end
puts i
```

Esse mesmo código na maioria das outras linguagens de programação retornaria algum tipo de erro. A variável `i` teria o escopo somente do bloco `for`. Em Ruby isso não acontece. Tome muito cuidado e lembre-se que o padrão é o das outras linguagens, em Ruby essa variável local é sempre local ao método como um todo.

9.3 Trapaceando

Mas queremos impressionar nossos amigos. Queremos ter certeza de que quando jogamos, ganhamos, por mais difícil que seja. Portanto, vamos verificar se quem joga sou eu mesmo:

```
if nome == seu nome
    ganhei!
end
```

Ótimo, para isso precisamos armazenar o nome. Sabemos que a função que dá boas-vindas ainda não retorna o nome, mas queremos:

```
nome = da_boas_vindas
```

Portanto nossa função passa a retornar o nome:

```
def da_boas_vindas
    puts "Bem vindo ao jogo da adivinhação"
    puts "Qual é o seu nome?"
    nome = gets.strip
    puts "\n\n\n\n\n\n"
    puts "Começaremos o jogo para você, #{nome}"
    nome
end
```

Ao rodar as tentativas, logo após ler o chute do jogador, verificamos se é você mesmo e, caso positivo, ganhamos:

```
for tentativa in 1..limite_de_tentativas
    chute = pede_um_numero chutes, tentativa,
            limite_de_tentativas
    chutes << chute

    if nome == "Guilherme"
        puts "Acertou!"
        break
    end

    # ...
end
```

Pronto! Já podemos mostrar o jogo para nossos amigos e o quão bom somos em adivinhação.

9.4 Corrigindo o número sorteado

Vimos que o array começa com a posição 0, a base binária e a hexadecimal (assim como a decimal) também. É justo por isso que o número aleatório também começa com zero e, consequentemente, nosso número secreto também. Mas isso é um tanto quanto contra intuitivo para o ser humano no dia a dia. Jogadores não estão acostumados a considerar o número zero em jogos de adivinhação, portanto vamos alterar nosso sorteio de número secreto para considerar somente entre `1` e o máximo, inclusive.

Arredondando:

```
def sorteia_numero_secreto(dificuldade)
    case dificuldade
    when 1
        maximo = 30
    when 2
        maximo = 60
    when 3
        maximo = 100
    when 4
        maximo = 150
    else
        maximo = 200
    end
    puts "Escolhendo um número secreto entre 1 e #{maximo}..."
    sorteado = rand(maximo) + 1
    puts "Escolhido... que tal adivinhar hoje nosso número
          secreto?"
    sorteado
end
```

Nosso código final fica:

```
def da_boas_vindas
    puts "Bem-vindo ao jogo da adivinhação"
    puts "Qual é o seu nome?"
    nome = gets.strip
    puts "\n\n\n\n\n\n"
    puts "Começaremos o jogo para você, #{nome}"
    nome
end

def pede_dificuldade
    puts "Qual o nível de dificuldade que deseja?
          (1 fácil, 5 difícil)"
    dificuldade = gets.to_i
end

def sorteia_numero_secreto(dificuldade)
    case dificuldade
    when 1
```

```
        maximo = 30
    when 2
        maximo = 60
    when 3
        maximo = 100
    when 4
        maximo = 150
    else
        maximo = 200
    end
    puts "Escolhendo um número secreto entre 1 e #{maximo}..."
    sorteado = rand(maximo) + 1
    puts "Escolhido... que tal adivinhar hoje nosso número
          secreto?"
    sorteado
end

def pede_um_numero(chutes, tentativa, limite_de_tentativas)
    puts "\n\n\n\n"
    puts "Tentativa #{tentativa} de #{limite_de_tentativas}"
    puts "Chutes até agora: #{chutes}"
    puts "Entre com o número"
    chute = gets.strip
    puts "Será que acertou? Você chutou #{chute}"
    chute.to_i
end

def verifica_se_acertou(numero_secreto, chute)
    acertou = numero_secreto == chute
    if acertou
        puts "Acertou!"
        return true
    end
    maior = numero_secreto > chute
    if maior
        puts "O número secreto é maior!"
    else
        puts "O número secreto é menor!"
    end
```

```
    false
end

nome = da_boas_vindas
dificuldade = pede_dificuldade
numero_secreto = sorteia_numero_secreto dificuldade

limite_de_tentativas = 5
chutes = []
pontos_ate_agora = 1000

for tentativa in 1..limite_de_tentativas
    chute = pede_um_numero chutes, tentativa,
            limite_de_tentativas
    chutes << chute

    if nome == "Guilherme"
        puts "Acertou!"
        break
    end

    pontos_a_perder = (chute - numero_secreto).abs / 2.0
    pontos_ate_agora = pontos_ate_agora - pontos_a_perder
    if verifica_se_acertou numero_secreto, chute
        break
    end
end

puts "Você ganhou #{pontos_ate_agora} pontos."
```

9.5 While: jogando diversas vezes

Jogar o jogo uma vez é divertido, mas jogar diversas vezes requer sua execução desde o começo, digitar nosso nome, escolher a dificuldade e poderia envolver ainda outros passos. Se desejamos executar nosso laço principal do jogo diversas vezes, nada mais natural do que extrair esse comportamento para uma função chamada `joga`. Pode reparar que ela precisa do `nome` e da `dificuldade`, que são as duas únicas variáveis independentes da rodada:

```
nome = da_boas_vindas
dificuldade = pede_dificuldade

joga nome, dificuldade
```

Nossa função extraída recebe os dois valores:

```
def joga(nome, dificuldade)
    numero_secreto = sorteia_numero_secreto dificuldade

    limite_de_tentativas = 5
    chutes = []
    pontos_ate_agora = 1000

    for tentativa in 1..limite_de_tentativas
        chute = pede_um_numero chutes, tentativa,
                limite_de_tentativas
        chutes << chute

        if nome == "Guilherme"
            puts "Acertou!"
            break
        end

        pontos_a_perder = (chute - numero_secreto).abs / 2.0
        pontos_ate_agora = pontos_ate_agora - pontos_a_perder
        if verifica_se_acertou numero_secreto, chute
            break
        end
    end

    puts "Você ganhou #{pontos_ate_agora} pontos."
end
```

Agora como fazer para jogar diversas vezes? Queremos executar o código `joga nome, dificuldade` eternamente. O laço de `for` executa uma mesma coisa um número determinado de vezes, que não é o que desejamos. Queremos eternamente, para sempre. Portanto, dizemos ao computador que `para sempre`:

```
while true
    joga nome, dificuldade
end
```

Claro, dizer `true` é algo estranho, o jogador um dia vai querer parar de jogar. Se criarmos uma função `quer_jogar`:

```
while quer_jogar
    joga nome, dificuldade
end
```

E definimos essa função, que pergunta se o usuário deseja jogar, retornando `true` se a resposta for `S`:

```
def quer_jogar
    puts "Deseja jogar novamente? (S/N)"
    quero_jogar = gets.strip
    quero_jogar == "S"
end
```

Mas ao tentarmos jogar, percebemos que ele para mesmo que eu digite `s`. O que acontece? Lembra de nossa tabela `ASCII`? Letra maiúscula não é igual à letra minúscula. Vamos invocar o método `upcase` de nossa `String`, que será capaz de transformá-la em maiúscula:

```
def quer_jogar
    puts "Deseja jogar novamente? (S/N)"
    quero_jogar = gets.strip
    quero_jogar.upcase == "S"
end
```

9.6 LOOP DO...END

Por mais que o jogo funcione, temos algo estranho. Logo após perguntar nosso nome e dificuldade, ele pergunta se desejamos jogar novamente:

```
Bem-vindo ao jogo da adivinhação
Qual é o seu nome?
guilherme
```

```
...
Começaremos o jogo para você, guilherme
Qual o nível de dificuldade que deseja? (1 fácil, 5 difícil)
1
...
Deseja jogar novamente? (S/N)
```

Não queremos executar o `quer_jogar` logo de cara. Mas a construção `while...end` primeiro executa a condição, e depois entra no laço. Queremos um laço, um `loop` que primeiro executa todo o código, e só quebra na condição específica de não querer jogar:

```
loop do
    joga nome, dificuldade
    if NAO quer_jogar
        break
    end
end
```

Mas como dizer que queremos inverter uma condição? Se for falsa, verdadeira; se for verdadeira, falsa? É o operador `!`:

```
loop do
    joga nome, dificuldade
    if !quer_jogar
        break
    end
end
```

Podemos escrever com o `if` em uma linha:

```
loop do
    joga nome, dificuldade
    break if !quer_jogar
end
```

Mas ainda está ruim. Misturamos um operador com português. Que tal renomearmos nossa função para `nao_quer_jogar`, parece ficar mais legível?

```
loop do
    joga nome, dificuldade
    break if nao_quer_jogar
end
```

Mais ainda se seguirmos o padrão do Ruby de utilizar `?` em funções e métodos que devolvem verdadeiro ou falso (`Booleans`):

```
loop do
    joga nome, dificuldade
    break if nao_quer_jogar?
end
```

Nossa função negativa vira:

```
def nao_quer_jogar?
    puts "Deseja jogar novamente? (S/N)"
    quero_jogar = gets.strip
    nao_quero_jogar = quero_jogar.upcase == "N"
end
```

Claro, assim como antes, se o usuário digitar alguma letra que não a esperada, o comportamento não é o ideal. Nosso código final é um jogo que podemos pedir para amigos e parentes jogarem, onde já utilizamos dezenas de conceitos de programação e computação:

```
def da_boas_vindas
    puts "Bem-vindo ao jogo da adivinhação"
    puts "Qual é o seu nome?"
    nome = gets.strip
    puts "\n\n\n\n\n\n"
    puts "Começaremos o jogo para você, #{nome}"
    nome
end

def pede_dificuldade
    puts "Qual o nível de dificuldade que deseja?
          (1 fácil, 5 difícil)"
    dificuldade = gets.to_i
end
```

```
def sorteia_numero_secreto(dificuldade)
    case dificuldade
    when 1
        maximo = 30
    when 2
        maximo = 60
    when 3
        maximo = 100
    when 4
        maximo = 150
    else
        maximo = 200
    end
    puts "Escolhendo um número secreto entre 1 e #{maximo}..."
    sorteado = rand(maximo) + 1
    puts "Escolhido... que tal adivinhar hoje nosso número
          secreto?"
    sorteado
end

def pede_um_numero(chutes, tentativa, limite_de_tentativas)
    puts "\n\n\n\n"
    puts "Tentativa #{tentativa} de #{limite_de_tentativas}"
    puts "Chutes até agora: #{chutes}"
    puts "Entre com o número"
    chute = gets.strip
    puts "Será que acertou? Você chutou #{chute}"
    chute.to_i
end

def verifica_se_acertou(numero_secreto, chute)
    acertou = numero_secreto == chute
    if acertou
        puts "Acertou!"
        return true
    end
    maior = numero_secreto > chute
    if maior
```

```
        puts "O número secreto é maior!"
    else
        puts "O número secreto é menor!"
    end
    false
end

def joga(nome, dificuldade)
    numero_secreto = sorteia_numero_secreto dificuldade

    limite_de_tentativas = 5
    chutes = []
    pontos_ate_agora = 1000

    for tentativa in 1..limite_de_tentativas
        chute = pede_um_numero chutes, tentativa,
                limite_de_tentativas
        chutes << chute

        if nome == "Guilherme"
            puts "Acertou!"
            break
        end

        pontos_a_perder = (chute - numero_secreto).abs / 2.0
        pontos_ate_agora = pontos_ate_agora - pontos_a_perder
        if verifica_se_acertou numero_secreto, chute
            break
        end
    end

    puts "Você ganhou #{pontos_ate_agora} pontos."
end

def nao_quer_jogar?
    puts "Deseja jogar novamente? (S/N)"
    quero_jogar = gets.strip
    nao_quero_jogar = quero_jogar.upcase == "N"
end
```

```
nome = da_boas_vindas
dificuldade = pede_dificuldade

loop
    joga nome, dificuldade
    break if nao_quer_jogar?
end
```

9.7 Resumindo

Vimos como refatorar uma sequência de `ifs` aninhados para um `case...when...end` quando as condições são aplicadas a uma única variável, lembrando que tais construções podem ser sensíveis a mudanças: se um novo valor é aceito, temos que lembrar de mudar todos os pontos da aplicação que fazem uso desses `ifs` e `cases`, o que é um cuidado que devemos tomar.

Entendemos melhor o escopo de variável local, criamos um recurso para um usuário determinado ganhar o jogo facilmente, corrigimos um problema com o número sorteado e aprendemos mais dois tipos de laço bastante utilizados: o `while...end` e o `loop do...end`.

Capítulo 10

Arte ASCII: jogo da adivinhação

10.1 Melhorando nossa interface com o usuário

Podemos deixar o nosso jogo mais atraente mudando a maneira como nos comunicamos com nosso usuário final. Um dos recursos mais antigos utilizados pelos programadores para construir uma interface divertida é criar imagens com caracteres comuns, do dia a dia. Na época em que estávamos restritos à tabela ASCII, usávamos somente os seus caracteres, e mesmo assim éramos capazes de fazer divertidas. Vamos alterar um pouco nossas funções para utilizar essa arte ASCII (*ASCII art*).

Primeiro alteremos nossa função `da_boas_vindas` para desenhar um castelo de boas-vindas:

```
def da_boas_vindas
```

```
  puts
  puts "        P  /_\  P                              "
  puts "       /_\_|_|_/_\                             "
  puts "   n_n | ||. .|| | n_n         Bem vindo ao    "
  puts "   |_|_|nnnn nnnn|_|_|     Jogo de Adivinhação!"
  puts "  |' '  |  |_|  |'  ' |                        "
  puts "  |_____| ' _ ' |_____|                        "
  puts "        \__|_|__/                              "
  puts
  puts "Qual é o seu nome?"
  nome = gets.strip
  puts "\n\n\n\n\n\n"
  puts "Começaremos o jogo para você, #{nome}"
  nome
end
```

Alteramos também a interface na hora de perguntar a dificuldade:

```
def pede_dificuldade
  puts "Qual o nível de dificuldade?"
  puts "(1) Muito fácil (2) Fácil (3) Normal (4) Difícil
        (5) Impossível"
  puts "Escolha: "
  dificuldade = gets.to_i
end
```

Ele começará a pedir palpites para o usuário. Enquanto o usuário erra, ele continua pedindo. Se você acertar, ficamos muito felizes. Extraímos uma função chamada `ganhou`:

```
def ganhou
  puts
  puts "             00000000000               "
  puts "         0000000000000000000           "
  puts "      000000  000000000  000000        "
  puts "    000000      00000      000000      "
  puts "  00000000  #   00000  #   00000000    "
  puts " 0000000000    0000000    0000000000   "
  puts "0000000000000000000000000000000000000  "
  puts "0000000000000000000000000000000000000  "
```

```
  puts "0000  0000000000000000000000000  0000   "
  puts " 0000  00000000000000000000000  0000    "
  puts "  0000   00000000000000000000  0000     "
  puts "    00000   000000000000000    0000     "
  puts "      000000   000000000    000000      "
  puts "         000000         000000          "
  puts "             000000000000               "
  puts
  puts "               Acertou!                 "
  puts
end
```

Nos momentos em que o jogador ganhou, invoque essa função. Testamos nosso jogo, agora sim temos muito mais diversão:

```
P  /_  P
       /__|_|_/_
   n_n | ||. .|| | n_n         Bem -vindo ao
   |_|_|nnnn nnnn|_|_|     Jogo de Adivinhação!
  |' '  |  |_|  |'  ' |
  |_____| ' _ ' |_____|
        __|_|__/


...
Qual o nível de dificuldade?
(1) Muito fácil (2) Fácil (3) Normal (4) Difícil (5) Impossível
...


             00000000000
         0000000000000000000
      000000  000000000  000000
    000000      00000      000000
  00000000  #   00000  #   00000000
 0000000000    0000000    0000000000
0000000000000000000000000000000000000
0000000000000000000000000000000000000
0000  0000000000000000000000000  0000
 0000  00000000000000000000000  0000
```

```
  0000   00000000000000000000  0000
    00000   000000000000000   0000
      000000   000000000   000000
         000000         000000
             000000000000

               Acertou!
...
```

Capítulo 11

Exercícios extras: jogo da adivinhação

Aqui você encontra exercícios extras para soltar sua imaginação e praticar lógica de programação na linguagem Ruby.

11.1 Melhorando o jogo de adivinhação

1) Se no último chute o usuário errar por um, avise-o que isso aconteceu e dê a ele uma última chance. Para isso você precisará verificar se ele está no último chute e errou por um, nesse caso aumente o número de tentativas em uma.

2) Temos ainda alguns números mágicos em nosso código. Extraia-os.

3) Não deixe o usuário jogar o mesmo número. Se ele jogou o número `2`, errou, e jogou novamente o `2`, avise que ele já jogou esse número, e não conte como uma tentativa.

4) Mostre quantas vezes você ganhou.

5) Mostre quantas vezes você perdeu.

6) Mostre a porcentagem de vezes que você ganhou. Cuidado com a divisão por `0`.

7) Ao perder o jogo mostre um ASCII art como:

```
\|/ ____ \|/
        @~/ ,. \~@
       /_( \__/ )_\
          \__U_/

Você perdeu! Tente novamente!
```

   Para isso você deve verificar se o usuário perdeu após o laço de tentativas terminar e somente nesse caso mostrar a mensagem de "perdeu". Lembre-se de extrair o código do `puts` em uma função separada, seu código ficará mais legível.

11.2 Outros desafios

1) Escreva um programa que imprima todos os números pares entre 2 e 50. Para saber se o número é par, basta você ver se o resto da divisão do número por 2 é igual a 0.

2) Escreva um programa que imprima a soma de todos os números de 1 até 100. Ou seja, ele calculará o resultado de 1+2+3+4+...+100.

3) Escreva um programa que peça um inteiro ao usuário, e com esse inteiro, ele imprima, linha a linha, a tabuada daquele número até o 10. Por exemplo, se ele escolher o número 2, o programa imprimirá: `2x1=2, 2x2=4, 2x3=6, ..., 2x10=20`.

4) Implemente uma calculadora. O programa deve pedir 3 números ao usuário, `a`, `b` e `operacao`. Se `operacao` for igual 1, você deverá imprimir a soma de `a + b`. Se ela for 2, a subtração. Se for 3, a multiplicação. Se for 4, a divisão.

5) Escreva um programa que peça um número inteiro ao usuário e imprima o fatorial desse número. Para calcular o fatorial, basta ir multiplicando pelos números anteriores até 1. Por exemplo, o fatorial de 4 é `4 * 3 * 2 * 1`, que é igual a 24.

Capítulo 12

Jogo da forca

Nosso próximo grande jogo é o da forca, onde o jogador deve adivinhar uma palavra, chutando cada vez uma nova letra ou a palavra inteira.

Começamos nosso arquivo `forca.rb` com a mensagem de boas-vindas de acordo com o nome do jogador:

```
def da_boas_vindas
    puts "Bem-vindo ao jogo da forca"
    puts "Qual é o seu nome?"
    nome = gets.strip
    puts "\n\n\n\n\n\n"
    puts "Começaremos o jogo para você, #{nome}"
    nome
end
```

O próximo passo é a escolha de uma palavra secreta. Queremos mostrar

o número de letras contidas na palavra secreta, para isso usamos o método `size`:

```
def sorteia_palavra_secreta
    puts "Escolhendo uma palavra..."
    palavra_secreta = "programador"
    puts "Escolhida uma palavra com #{palavra_secreta.size}
          letras... boa sorte!"
    palavra_secreta
end
```

Definimos também a função que pergunta se o jogador deseja jogar novamente:

```
def nao_quer_jogar?
    puts "Deseja jogar novamente? (S/N)"
    quero_jogar = gets.strip
    nao_quero_jogar = quero_jogar.upcase == "N"
end
```

O nosso laço principal jogará o jogo diversas vezes:

```
nome = da_boas_vindas

loop do
    joga nome
    break if nao_quer_jogar?
end
```

Agora a única grande diferença está em nosso `joga`. Nossa função de jogar deve escolher a palavra secreta, configurar o erro até agora e começar com zero ponto.

```
def joga(nome)
    palavra_secreta = sorteia_palavra_secreta

    erros = 0
    chutes = []
    pontos_ate_agora = 0
```

```
    # laco principal a escrever

    puts "Você ganhou #{pontos_ate_agora} pontos."
end
```

O laço principal deve ser executado até o número de erros chegar a 5, isto é, enquanto `erros` for menor que `5`:

```
def joga(nome)
    palavra_secreta = sorteia_palavra_secreta

    erros = 5
    chutes = []
    pontos_ate_agora = 0

    while erros < 5
        # o jogo
    end

    puts "Você ganhou #{pontos_ate_agora} pontos."
end
```

Agora definimos a função que lê a palavra chutada. Mas, diferentemente de antes, não temos um limite de tentativas e não armazenamos a tentativa atual, somente os chutes e o número de erros:

```
def pede_um_chute(chutes, erros)
    puts "\n\n\n\n"
    puts "Erros até agora: #{erros}"
    puts "Chutes até agora: #{chutes}"
    puts "Entre com a letra ou palavra"
    chute = gets.strip
    puts "Será que acertou? Você chutou #{chute}"
    chute
end
```

Podemos continuar com nosso laço principal, pedindo o chute e guardando-o no array de chutes:

```
def joga(nome)
    palavra_secreta = sorteia_palavra_secreta

    erros = 0
    chutes = []
    pontos_ate_agora = 0

    while erros < 5
        chute = pede_um_chute chutes, erros
        chutes << chute

        # colocar as regras de acerto e erro aqui!

    end

    puts "Você ganhou #{pontos_ate_agora} pontos."
end
```

12.1 Chute de uma palavra completa e a comparação com ==

O jogador ganha 100 pontos a cada palavra chutada certa e perde 30 pontos a cada palavra inteira chutada errada. Ele não perde nem ganha pontos ao chutar uma letra, afinal com cinco chutes errados ele perde o jogo.

Falta colocarmos as regras de acerto e erro. Primeiro devemos conferir se o usuário chutou uma única letra, isto é, se sua entrada tem tamanho 1, ou se o chute foi da palavra inteira:

```
if chute.size == 1
    # chutou uma única letra
else
    # chutou a palavra inteira
end
```

Mas já falamos qual o problema de comentários. Vamos= extrair uma variável, algo que nos diga se ele `chutou_uma_unica_letra`:

```
chutou_uma_unica_letra = chute.size == 1
if chutou_uma_unica_letra
```

```
else

end
```

Caso ele tenha chutado a palavra inteira, e acertado, mostramos a mensagem de parabéns e paramos a jogada com o `break` que já conhecemos:

```
chutou_uma_unica_letra = chute.size == 1
if chutou_uma_unica_letra

else
    acertou = chute == palavra_secreta
    if acertou
        puts "Parabéns! Acertou!"
        pontos_ate_agora += 100
        break
    else
    end
end
```

Caso ele erre, tiramos os `30` pontos e aumentamos um erro:

```
chutou_uma_unica_letra = chute.size == 1
if chutou_uma_unica_letra

else
    acertou = chute == palavra_secreta
    if acertou
        puts "Parabéns! Acertou!"
        pontos_ate_agora += 100
        break
    else
        puts "Que pena... errou!"
        pontos_ate_agora -= 30
        erros += 1
    end
end
```

Nosso jogo já funciona para chutes inteiros da palavra, por exemplo se eu

tentar chutar a palavra *programadores* (-30 pontos), e depois a palavra *programador* (+100 pontos), o jogo para com meus 70 pontos:

```
Erros até agora: 0
Chutes até agora: []
Entre com a letra ou palavra
programadores
Será que acertou? Você chutou programadores
Que pena... errou!
...
Erros até agora: 1
Chutes até agora: ["programadores"]
Entre com a letra ou palavra
programador
Será que acertou? Você chutou programador
Parabéns! Acertou!
Você ganhou 70 pontos.
...
```

12.2 Encontrando um algoritmo

Mas precisamos agora suportar os chutes de letras. Cada vez que o jogador tenta uma única letra devemos verificar se a nossa `String palavra_secreta` inclui o caractere escolhido. Seria interessante dizer quantas vezes encontramos tal letra. Se quisermos contar quantas vezes encontramos uma letra em uma palavra, como poderemos fazer? Vamos pensar como fazemos isso como crianças. Esse é o segredo para escrever em código a solução de qualquer problema lógico: lembre-se como você resolve o problema, e implemente-o. Mas cuidado, lembre-se como uma criança, pois a criança usa as estruturas mais básicas de lógica (ela não usa potência, nem log, nem coisas complexas, somente `ifs` e `fors`).

Por exemplo, dada a palavra mágica, `programador` para uma criança, como ela conta quantas vezes aparece a letra `o`? Ela escreve zero em um cantinho do papel (ou na cabeça, na memória!). Depois ela olha a primeira letra da palavra (aponta para ela com o dedo!). Essa letra é a letra `o` que estou procurando? Não. Vai para a próxima. É? Não. Próxima. É? Sim! Então soma um no cantinho do papel (na memória). Continua até o fim.

No fim, se o número é zero, significa que a letra não está lá dentro, aumenta um erro. Se o número é diferente de zero, ele representa quantas vezes a letra foi encontrada. Pronto. Temos nossa estrutura lógica para resolver o problema, temos nosso algoritmo.

Estou usando o exemplo de uma criança, mas não se assuste, mesmo que vá resolver um problema de grafos complexos em que deseja encontrar a menor e mais barata rota entre três cidades via avião a ideia é ter em mente que descrever o problema em português, e como você (um adulto, uma criança grande, não importa) o resolveria com estruturas básicas de lógica é a chave para escrever o algoritmo na linguagem de programação que deseja. Com o passar do tempo, claro, você começa a eliminar esse passo. Não tenha pressa nenhuma. Dois meses, seis meses, seis anos. Os programadores que você já viu escrevendo código têm anos de experiência e por isso já fizeram o mesmo que você está fazendo milhares de vezes. Com milhares de pontos de XP você também será capaz de escrever o algoritmo de contar letras de cabeça.

12.3 Implementando o algoritmo

Tomemos a estrutura *o algoritmo* ao pé da letra, como foi descrita aqui:

```
dada a palavra mágica, %%programador%% para uma criança,
Ela escreve zero em um cantinho do papel (ou na cabeça, na
memória!).
Depois ela olha a primeira letra da palavra (aponta para ela com
o dedo!).
Essa letra é a letra %%o%% que estou procurando? Não.
Vai para a próxima.
É? Não.
Próxima.
É? Sim!
Então soma um no cantinho do papel (na memória).
Continua até o fim.

No fim,
se o número é zero,
significa que a letra não está lá dentro, aumenta um erro
se o número é diferente de zero,
```

```
ele representa quantas vezes a letra foi encontrada.
```

Agora temos que traduzir nossa estrutura para Ruby. Sem problemas. Comecemos com as variáveis:

```
palavra_secreta = "programador"
chute = "o"
total_encontrado = 0

Depois ela olha a primeira letra da palavra (aponta para ela com
o dedo!).
Essa letra é a letra %%o%% que estou procurando? Não.
Vai para a próxima.
É? Não.
Próxima.
É? Sim!
Então soma um no cantinho do papel (na memória).
Continua até o fim.

No fim,
se o número é zero,
significa que a letra não está lá dentro, aumenta um erro
se o número é diferente de zero,
ele representa quantas vezes a letra foi encontrada.
```

Agora precisamos traduzir nosso laço, quando a criança passa para cada letra:

```
palavra_secreta = "programador"
chute = "o"
total_encontrado = 0

for i = 0..(palavra_secreta.size - 1)
    letra = palavra_secreta[i]
    Essa letra é a letra %%o%% que estou procurando? Não.
    É? Não.
    É? Sim!
    Então soma um no cantinho do papel (na memória).
end
```

```
No fim,
se o número é zero,
significa que a letra não está lá dentro, aumenta um erro
se o número é diferente de zero,
ele representa quantas vezes a letra foi encontrada.
```

Se a letra for a letra correta, devemos atualizar nosso contador:

```
palavra_secreta = "programador"
chute = "o"
total_encontrado = 0

for i = 0..(palavra_secreta.size - 1)
    letra = palavra_secreta[i]
    if letra == chute
        total_encontrado += 1
    end
end
```

```
No fim,
se o número é zero,
significa que a letra não está lá dentro, aumenta um erro
se o número é diferente de zero,
ele representa quantas vezes a letra foi encontrada.
```

E no fim mostrar o resultado:

```
palavra_secreta = "programador"
chute = "o"
total_encontrado = 0

for i = 0..(palavra_secreta.size - 1)
    letra = palavra_secreta[i]
    if letra == chute
        total_encontrado += 1
    end
end

if total_encontrado == 0
    puts "Letra não encontrada!"
```

```
    erros += 1
else
    puts "Letra encontrada #{total_encontrado} vezes!"
end
```

Claro, no nosso código já temos a variável `palavra_secreta` e `chute`. Portanto, nossa função `joga` fica:

```
def joga(nome)
    palavra_secreta = sorteia_palavra_secreta

    erros = 0
    chutes = []
    pontos_ate_agora = 0

    while erros < 5
        chute = pede_um_chute chutes, erros
        chutes << chute

        chutou_uma_unica_letra = chute.size == 1
        if chutou_uma_unica_letra
            total_encontrado = 0

            for i = 0..(palavra_secreta.size - 1)
                letra = palavra_secreta[i]
                if letra == chute
                    total_encontrado += 1
                end
            end

            if total_encontrado == 0
                puts "Letra não encontrada!"
                erros += 1
            else
                puts "Letra encontrada #{total_encontrado}
                      vezes!"
            end
        else
            acertou = chute == palavra_secreta
```

```
            if acertou
                puts "Parabéns! Acertou!"
                pontos_ate_agora += 100
                break
            else
                puts "Que pena... errou!"
                pontos_ate_agora -= 30
            end
        end

    end

    puts "Você ganhou #{pontos_ate_agora} pontos."
end
```

Esse código é uma "bela" de uma função. Um codigozão. Em tamanho, não em qualidade. Pelo contrário, é assustador o tamanho, e o que fazemos com código horrível que o Guilherme escreve? Refatoramos.

12.4 Boa prática: explorando a documentação

Novamente vamos aplicar o conceito de extrair um trecho de código, uma função nova. Repare que o ato de contar quantas vezes um caractere aparece em uma `String` parece ser algo razoavelmente isolável. Tire ele dali:

```
def conta(texto, caracter)
    total_encontrado = 0

    for i = 0..texto.size
        letra = texto[i]
        if letra == caracter
            total_encontrado += 1
        end
    end

    total_encontrado
end
```

Mas se olharmos na documentação, `String`s possuem um método chamado `chars` que devolve um array de caracteres, que facilita nosso laço:

```
def conta(texto, caracter)
    total_encontrado = 0

    for letra in texto.chars
        if letra == caracter
            total_encontrado += 1
        end
    end

    total_encontrado
end
```

Ótimo, o código ficou mais simples. Mas já que estamos olhando a documentação, é importante olharmos com calma, o que é aquele método `count`? Ele conta o número de aparições de um caractere em uma `String` (entre outras utilizações possíveis). Logo, podemos remover completamente nossa nova função! Esse código já existe no Ruby!

Mas Guilherme, porque você me levou a criar toda essa função, com um laço, um contador, para depois refatorar e por fim olhar a documentação e perceber que o código já existia?

Não estamos aqui aprendendo a ser *copy e pasters*. Queremos entender como um programa funciona e o que deve ser feito para se resolver um problema. Sempre que você usar uma biblioteca ou função é importante entender como ela resolve seu problema. Podemos não saber os mínimos detalhes, mas se não entendermos o que acontece por trás, a chance de algum bug acontecer é maior, afinal ela pode fazer algo que não sabemos que ela faz.

Nesse caso específico, é vital para um programador entender como funciona um laço com um contador, esse tipo de estrutura lógica é usado em todo canto quando programamos dando comandos, ordens (programação imperativa) como fizemos até agora. Desde o algoritmo comercial do cálculo do total de uma compra até um algoritmo matemático encontrar os fatores primos de um número qualquer, laços com acumuladores (números ou arrays) aparecem e reaparecem.

Como regra geral, sempre que quisermos fazer algo com um valor do tipo `X` (por exemplo `String` ou `Fixnum`), vale *muito*, *muito*, *muito* a pena olhar a documentação e ver se já existe um método que faça isso.

E conforme você já deve ter percebido, sempre veremos como as coisas funcionam, para então melhorar nosso código.

Como fica então nossa função `joga`?

```
def joga(nome)
    palavra_secreta = sorteia_palavra_secreta

    erros = 0
    chutes = []
    pontos_ate_agora = 0

    while erros < 5
        chute = pede_um_chute chutes, tentativa,
                limite_de_tentativas
        chutes << chute

        chutou_uma_unica_letra = chute.size == 1
        if chutou_uma_unica_letra
            total_encontrado = palavra_secreta.count(chute[0])
            if total_encontrado == 0
                puts "Letra não encontrada!"
                erros += 1
            else
                puts "Letra encontrada #{total_encontrado}
                vezes!"
            end
        else
            acertou = chute == palavra_secreta
            if acertou
                puts "Parabéns! Acertou!"
                pontos_ate_agora += 100
                break
            else
                puts "Que pena... errou!"
                pontos_ate_agora -= 30
                erros += 1
```

```
            end
        end

    end

    puts "Você ganhou #{pontos_ate_agora} pontos."
end
```

12.5 next... Evitando chutes repetidos

Por mais que nosso jogo funcione, tem algo muito estranho ainda, o jogo permite que eu tente duas vezes a mesma letra ou palavra, algo que não faz sentido. Ele poderia me avisar. Como fazer isso?

Pensando como criança, para saber se eu já falei uma letra ou palavra, devo toda vez que falar uma letra ou uma palavra anotá-la. Depois quando falo uma letra ou palavra nova basta conferir se ela já foi anotada antes.

Já temos as palavras anotadas em um array, o `chutes`. Basta verificarmos:

```
chute = pede_um_chute chutes, tentativa,
        limite_de_tentativas
# verificar aqui se ja chutei
chutes << chute
```

Como verificar se um valor já está presente em um array? Como criança, tenho que ir para todos os elementos do array, comparando um a um, até encontrar. Se encontrar um igual, verdadeiro, tem. Se nenhum for igual, não encontrei, não tem. Isso é, um laço com acumulador (ou *early return*), como de costume. Em vez de implementarmos isso, parece razoável imaginar que essa tarefa seja tão comum que a linguagem nos forneça isso. Sim, ela provê uma função chamada `include?` que nos diz se o array inclui o valor que procuramos.

```
chute = pede_um_chute chutes, tentativa,
        limite_de_tentativas
if chutes.include? chute
    puts "Você já chutou #{chute}"
```

```
end
chutes << chute
```

Tentamos e o que acontece?

```
Entre com a letra ou palavra
p
...
Chutes até agora: ["p"]
Entre com a letra ou palavra
p
Será que acertou? Você chutou p
Você já chutou p
Letra encontrada 1 vezes!
...
Chutes até agora: ["p", "p"]
Entre com a letra ou palavra
```

Ele não ignorou nossa segunda letra, adicionou no array uma segunda vez, apesar de mostrar a mensagem que já a havíamos pedido. Claro, no nosso `if` nós somente mostramos a mensagem, precisamos escrever o código que continua nosso laço, ignorando todo o resto do código. Mas como fazer isso?

Precisamos de algum tipo de código que seja capaz de continuar o laço, ir para a próxima rodada, a próxima iteração, sem que ele pare. A palavra `break` quebrava o laço, não é o que queremos. Queremos continuar na próxima rodada, *next* (em diversas linguagens, *continue*).

```
while erros < 5
    chute = pede_um_chute chutes, tentativa,
            limite_de_tentativas
    if chutes.include? chute
        puts "Você já chutou #{chute}"
        next
    end
    chutes << chute

    ...
end
```

Agora sim, ao repetirmos uma letra, ela não é adicionada ao array novamente, nem verificada:

```
...
p
Será que acertou? Você chutou p
Letra encontrada 1 vezes!
...
Erros até agora: 0
Chutes até agora: ["p"]
Entre com a letra ou palavra
p
Será que acertou? Você chutou p
Você já chutou p
...
```

12.6 Resumindo

Vimos como utilizar diversas características que aprendemos anteriormente de uma linguagem de programação, como laços (`while`, `for`, `loop`), condicionais (`if`), entrada e saída (`puts` e `gets`), além de usar o `strip`, `arrays`, soma e subtração.

Aprendemos a utilização do `break` e do `next` para quebrar ou continuar um laço, o `count` para contar elementos e onde encontrar a documentação da linguagem para evitar reinventar a roda.

Entendemos também como funciona o coração de um programa: a criação e descrição de um processo que resolve um problema, seu algoritmo. Pensando como crianças somos capazes de quebrar o algoritmo em partes tão pequenas que condicionais e laços podem descrevê-lo, portanto criando nosso código a partir dessa descrição.

Capítulo 13

Responsabilidades

13.1 Mostrando parte da palavra secreta

Nosso jogo ainda é difícil de ganhar: não mostramos as posições onde as letras foram encontradas. Por exemplo, ao chutar a letra `o` deveríamos ter a resposta do programa:

```
__o______o_
```

Depois de `o`, ao chutar `a`:

```
__o__a_a_o_
```

Como podemos fazer isso? Primeiro pensamos onde desejamos implementar esse código. Vamos dar essa informação como resposta ao jogador, um *feedback* a cada nova rodada. Vamos imprimi-la durante o processo de pedir um novo chute, dentro do `pede_um_chute`:

```
def pede_um_chute(chutes, erros)
    puts "\n\n\n\n"
    puts "Erros até agora: #{erros}"
    puts "Chutes até agora: #{chutes}"
    puts "Entre com a letra ou palavra"
    chute = gets.strip
    puts "Será que acertou? Você chutou #{chute}"
    chute
end
```

Repare que a função de `pedir_um_chute` é uma função de interface com o jogador, com o usuário (*user interface* ou *UI*). Diferente de um código como a função `joga`, que é de lógica do nosso jogo, lógica do nosso negócio (*business logic*).

Tudo em um arquivo, 80 linhas, tudo como se fosse um escopo global, qualquer um pode fazer qualquer coisa. De onde venho, chamamos isso de "mistura", tem um pouco de tudo, junta tudo num único recipiente. Na comida fica gostoso, mas aqui fica ruim de entender.

Chegou a hora de começarmos a separar quem é quem, o que é o quê. E todo tipo de separação é feita em níveis. Já fizemos uma primeira, quando extraímos e isolamos variáveis; uma segunda, quando extraímos e isolamos funções; agora vamos extrair e isolar partes de nosso jogo de acordo com suas responsabilidades. Queremos uma única responsabilidade por arquivo, para ficar mais fácil de encontrar o que está em que lugar.

13.2 Separando a interface com o usuário da lógica de negócios

As funções que temos em nosso programa são:

```
da_boas_vindas
sorteia_palavra_secreta
pede_um_chute
nao_quer_jogar?
joga
```

Vale lembrar que nosso jogo possui uma interface de texto com o usuário, tanto para a entrada quanto para a saída. Dessas funções, `da_boas_vindas`, `pede_um_chute` e `nao_quer_jogar?` são funções claramente de interface com o usuário, elas mostram e pedem informação, sem executar nenhum código de lógica. São somente sequências de invocações a funções do tipo `gets` e `puts`, no máximo com uma conversão de um dado de entrada que é `String` para um `booleano` (verdadeiro ou falso).

Já a função `sorteia_palavra_secreta` por enquanto só efetua `puts`, não possui nenhuma lógica, portanto a colocaremos também no grupo de interface com o usuário.

Vamos criar um arquivo chamado `ui.rb` e colocamos nele nossas funções:

```
def da_boas_vindas
    puts "Bem-vindo ao jogo da forca"
    puts "Qual é o seu nome?"
    nome = gets.strip
    puts "\n\n\n\n\n\n"
    puts "Começaremos o jogo para você, #{nome}"
    nome
end

def pede_um_chute(chutes, erros)
    puts "\n\n\n\n"
    puts "Erros até agora: #{erros}"
    puts "Chutes até agora: #{chutes}"
    puts "Entre com a letra ou palavra"
    chute = gets.strip
    puts "Será que acertou? Você chutou #{chute}"
    chute
end

def nao_quer_jogar?
    puts "Deseja jogar novamente? (S/N)"
    quero_jogar = gets.strip
    nao_quero_jogar = quero_jogar.upcase == "N"
end
```

```
def sorteia_palavra_secreta
    puts "Escolhendo uma palavra..."
    palavra_secreta = "programador"
    puts "Escolhida uma palavra com #{palavra_secreta.size}
          letras... boa sorte!"
    palavra_secreta
end
```

É fácil notar que a última função, `joga`, é na verdade uma mistura de lógica (`ifs`, `laços` etc.) e interface com o usuário (`puts`). O que fazer com ela? Vamos analisar caso a caso cada um dos `puts` de nossa função, extraindo a parte de `UI` da de lógica, para desembaralhar o spaguetti que temos.

Primeiro encontramos o código que verifica se já chutamos esse mesmo valor anteriormente:

```
chute = pede_um_chute chutes, erros
if chutes.include? chute
    puts "Você já chutou #{chute}"
    next
end
chutes << chute
```

O que podemos fazer aqui? Uma solução é criar uma função chamada `avisa_chute_repetido`, que somente invoca o `puts`, no nosso arquivo `ui.rb`:

```
def avisa_chute_repetido(chute)
    puts "Você já chutou #{chute}"
end
```

E alterar nosso código de lógica de valor inválido para chamar essa função:

```
chute = pede_um_chute chutes, erros
if chutes.include? chute
    avisa_chute_repetido chute
    next
end
chutes << chute
```

Depois temos a situação em que procuramos a letra, com dois possíveis resultados impressos:

```
if total_encontrado == 0
    puts "Letra não encontrada!"
    erros += 1
else
    puts "Letra encontrada #{total_encontrado}
          vezes!"
end
```

Trocamos por novos métodos, de responsabilidade bem clara em nosso `ui.rb`:

```
def avisa_letra_nao_encontrada
    puts "Letra não encontrada!"
end
def avisa_letra_encontrada(total_encontrado)
    puts "Letra encontrada #{total_encontrado} vezes!"
end
```

Os dois casos de chute de palavra completa também podem ser extraídos em funções para o nosso `ui.rb`:

```
def avisa_acertou_palavra
    puts "Parabéns! Acertou!"
end
def avisa_errou_palavra
    puts "Que pena... errou!"
end
```

Nosso último `puts` da função `joga` é aquele que mostra os pontos:

```
def avisa_pontos(pontos_ate_agora)
    puts "Você ganhou #{pontos_ate_agora} pontos."
end
```

Ficamos com a função `joga` limpa de qualquer invocação direta a `puts` ou `gets`. Tudo o que é feito com a interface do usuário está feito no arquivo `ui.rb`:

```ruby
def joga(nome)
    palavra_secreta = sorteia_palavra_secreta

    erros = 0
    chutes = []
    pontos_ate_agora = 0

    while erros < 5
        chute = pede_um_chute chutes, erros
        if chutes.include? chute
            avisa_chute_repetido chute
            next
        end
        chutes << chute

        chutou_uma_unica_letra = chute.size == 1
        if chutou_uma_unica_letra
            total_encontrado = palavra_secreta.count(chute[0])
            if total_encontrado == 0
                avisa_letra_nao_encontrada
                erros += 1
            else
                avisa_letra_encontrada total_encontrado
            end
        else
            acertou = chute == palavra_secreta
            if acertou
                avisa_acertou_palavra
                pontos_ate_agora += 100
                break
            else
                avisa_errou_palavra
                pontos_ate_agora -= 30
                erros += 1
            end
        end

    end
```

```
    avisa_pontos pontos_ate_agora
end
```

Portanto, nosso arquivo `forca.rb` possui a definição da função `joga` e a invocação do jogo:

```
nome = da_boas_vindas

loop do
    joga nome
    break if nao_quer_jogar?
end
```

Testamos rodar nosso `ruby forca.rb` e temos um erro:

```
forca.rb:43:in `<main>': undefined local variable or method
    `da_boas_vindas' for main:Object (NameError)
```

Ele não encontra a função `da_boas_vindas` do nosso outro arquivo, `ui.rb`. Claro, um arquivo não vê outro arquivo por padrão. Seria uma loucura que o interpretador abrisse todos os arquivos de nosso computador automaticamente, por padrão, para buscar tudo o que fosse necessário. Por isso é necessário indicar ao `forca.rb` que precisamos do arquivo de interface com o usuário, requeremos o carregamento do arquivo `ui.rb` relativo ao diretório atual, indicando logo no começo de nosso `forca.rb`:

```
require_relative 'ui'

def joga(nome)
    # ...
end

# ...
```

Agora sim, rodamos nosso jogo e tudo volta a funcionar normalmente!

REQUIRE

Assim como existe a instrução `require_relative`, há também a `require`. Ela busca em uma lista de diretórios específicos pelo arquivo mencionado. Por exemplo, podemos usá-la para carregar a biblioteca padrão de acesso HTTP:

```
require "net/http"
require "uri"

uri = URI.parse("http://www.casadocodigo.com.br")
Net::HTTP.get_print(uri)
```

13.3 Extraindo a lógica de negócios

Nossa função `joga` é a nossa lógica de negócios principal e claramente grande demais. Ela é mantida no arquivo chamado `forca.rb`. Mas também, pensando assim, o início de nosso jogo é um código de lógica:

```
nome = da_boas_vindas

loop do
    joga nome
    break if nao_quer_jogar?
end
```

Nada mais natural nessa visão do que isolá-lo em uma função em nosso `forca.rb` que se chame `jogo_da_forca`:

```
require_relative 'ui'

def joga(nome)
    # ...
end

def jogo_da_forca
```

```
    nome = da_boas_vindas

    loop do
        joga nome
        break if nao_quer_jogar?
    end
end
```

Terminamos nosso jogo nesse instante com dois arquivos. O `forca.rb` com as duas funções de lógica, o `ui.rb` com nossa função de interface com o usuário. Mas e a chamada para o `jogo_da_forca`? Devemos colocá-la em algum lugar. Vamos criar o arquivo `main.rb` com a invocação ao início do jogo:

```
require_relative 'logic'

jogo_da_forca
```

Note que todo programa costuma ter uma função de entrada. Nosso código não está mais voando, como código global. Isso é tão comum que em diversas linguagens é obrigatória a definição dessa função principal (*main*).

13.4 Extraindo a lógica de um chute válido

Desejamos fazer uma pequena extração de função para ver como ficou mais fácil entender onde está nosso código e onde ele deve ficar. Pense na lógica de não permitir chutes repetidos. Encontrou?

```
chute = pede_um_chute chutes, erros
if chutes.include? chute
    avisa_chute_repetido chute
    next
end
```

Que código feio, note quanta informação em cinco linhas. Isso no meio de uma função de cerca de 40 linhas. Vamos isolá-lo em uma lógica clara, que pede um chute *válido*:

```
chute = pede_um_chute_valido chutes, erros
```

Agora basta definir a função de `pede_um_chute_valido`. Mas calma, onde deve ficar essa função de lógica, que chama nossa UI diversas vezes (até um chute válido)? No arquivo de lógica, claro. Nossa função faz um loop e pede um chute até que ele seja válido:

```
def pede_um_chute_valido(chutes, erros)
    loop do
        chute = pede_um_chute chutes, erros
        if chutes.include? chute
            avisa_chute_repetido chute
        else
            return chute
        end
    end
end
```

Repare que, ao pedir novamente um chute, imprimimos novas quatro linhas e todo o cabeçalho, meio repetitivo. Vamos mudar nossas duas funções que pedem um chute para simplificá-las. Ao pedir o chute mostramos somente uma vez as linhas em branco, os erros e os chutes:

```
# no arquivo ui.rb
def cabecalho_de_tentativa(chutes, erros)
    puts "\n\n\n\n"
    puts "Erros até agora: #{erros}"
    puts "Chutes até agora: #{chutes}"
end

# no arquivo forca.rb
def pede_um_chute_valido(chutes, erros)
    cabecalho_de_tentativa chutes, erros
    loop do
        chute = pede_um_chute chutes, erros
        if chutes.include? chute
            avisa_chute_repetido chute
        else
            return chute
        end
```

```
    end
end
```

Pedir um chute fica mais simples:

```
def pede_um_chute(chutes, erros)
    puts "Entre com a letra ou palavra"
    chute = gets.strip
    puts "Será que acertou? Você chutou #{chute}"
    chute
end
```

Podemos inclusive remover os dois parâmetros ao pedir um chute:

```
def pede_um_chute(chutes, erros)
    puts "Entre com a letra ou palavra"
    chute = gets.strip
    puts "Será que acertou? Você chutou #{chute}"
    chute
end
```

Claro, ao invocar a função não precisamos mais dos parâmetros:

```
chute = pede_um_chute
```

Ficamos com as duas funções de `ui`:

```
def pede_um_chute
    puts "Entre com a letra ou palavra"
    chute = gets.strip
    puts "Será que acertou? Você chutou #{chute}"
    chute
end

def cabecalho_de_tentativa(chutes, erros)
    puts "\n\n\n\n"
    puts "Erros até agora: #{erros}"
    puts "Chutes até agora: #{chutes}"
end
```

E uma de lógica:

```
def pede_um_chute_valido(chutes, erros)
    cabecalho_de_tentativa chutes, erros
    loop do
        chute = pede_um_chute
        if chutes.include? chute
            avisa_chute_repetido chute
        else
            return chute
        end
    end
end
```

13.5 Implementação: mostrando parte da palavra secreta

Agora que isolamos duas camadas lógica de negócio e interface com o usuário estamos preparados para refatorar e adicionar novas funcionalidades com mais facilidade e organização.

Nosso próximo passo é implementar o mostrador de palavra parcial como citamos anteriormente. Por exemplo, ao chutar as letras `a` e `o`, teríamos:

```
__o__a_a_o_
```

Qual o algoritmo para fazer isso? Queremos imprimir cada letra de nossa palavra secreta, logo temos que ver se cada uma delas já foi chutada. Se sim, usa a letra, se não, usa o `_` (*underline*).

Temos nosso algoritmo! Repare que foram utilizadas somente as palavras *para cada* (*laço*) e *se/se não* (*if/else*):

```
para cada letra in palavra_secreta
    if letra ja foi chutada
        usa letra
    else
        usa "_"
    end
end
```

Claro, vamos traduzir para Ruby:

```
for letra in palavra_secreta.chars
    if chutes.include? letra
        puts letra
    else
        puts "_"
    end
end
```

Esse código é uma função de lógica misturada com interface, não parece ser bom. Além disso, cada `puts` vai imprimir uma quebra de linha, teremos uma letra por linha, que não é o que queremos. Em vez disso, podemos criar uma única função de lógica, `palavra_mascarada`, que, dado os chutes e a palavra secreta, devolve a palavra mascarada:

```
def palavra_mascarada(chutes, palavra_secreta)
    mascara = ""
    for letra in palavra_secreta.chars
        if chutes.include? letra
            mascara += letra
        else
            mascara += "_"
        end
    end
    mascara
end
```

Essa palavra pode ser calculada logo antes de pedir um chute, no começo do laço:

```
# ...
    while erros < 5
        mascara = palavra_mascarada chutes, palavra_secreta
        chute = pede_um_chute_valido chutes, erros
        chutes << chute
# ...
```

Ao pedir o chute, passaremos a máscara também como parâmetro para a interface com o usuário:

```
# ...
    while erros < 5
        mascara = palavra_mascarada chutes, palavra_secreta
        chute = pede_um_chute_valido chutes, erros, mascara
        chutes << chute
# ...
```

Vamos agora à interface com o usuário para fazer com que, ao pedir um chute válido, mostremos a mensagem com a máscara:

```
# ui.rb
def cabecalho_de_tentativa(chutes, erros, mascara)
    puts "\n\n\n\n"
    puts "Palavra secreta: #{mascara}"
    puts "Erros até agora: #{erros}"
    puts "Chutes até agora: #{chutes}"
end

# forca.rb
def pede_um_chute_valido(chutes, erros, mascara)
    cabecalho_de_tentativa chutes, erros, mascara
    loop do
        chute = pede_um_chute
        if chutes.include? chute
            avisa_chute_repetido chute
        else
            return chute
        end
    end
end
```

Agora sim, jogamos uma rodada, escolhendo a letra `a`:

```
Palavra secreta: ___________
...
Será que acertou? Você chutou a
Letra encontrada 2 vezes!
...
Palavra secreta: _____a_a___
Erros até agora: 0
```

```
Chutes até agora: ["a"]
...
```

Note que ao adicionar uma nova funcionalidade alteramos o arquivo de lógica, nosso `forca.rb`. Ao alterar a maneira como o programa interage com o usuário, mudamos o `ui.rb`. Se adicionamos uma nova funcionalidade que também muda a iteração com o usuário, alteramos ambos os arquivos.

13.6 Resumindo

Aprendemos a separar o código de lógica de negócios do código de interface com o usuário (*UI*) e para deixar tal separação ainda mais óbvia criamos arquivos distintos, gerando a dependência (o requerimento) entre eles através do `require_relative`.

Ao finalmente adicionar a lógica de mostrar a palavra secreta com os chutes nas posições adequadas fomos capazes de perceber que a estrutura criada nos ajuda a definir onde vai cada modificação: uma nova lógica de negócios e uma nova iteração através da interface com o usuário vão em arquivos distintos.

Em sistemas maiores ou utilizando práticas de controle de escopo (*namespace*), de orientação a objetos (*OO*) ou linguagens funcionais, somos capazes de organizar ainda mais nosso código. À medida que evoluir no aprendizado de linguagens você terá a oportunidade de aprender cada um desses conceitos, tudo no momento adequado e dependendo do caminho que escolher para seu próximo passo.

Capítulo 14

Entrada e saída de arquivo: palavras aleatórias e o top player

14.1 Lendo um arquivo de palavras, nosso dicionário

Chegamos aos nossos últimos desafios do jogo da forca. Primeiro queremos fazer com que a lista de palavras seja lida de um arquivo, para que o jogador não tenha ideia de qual palavra estamos utilizando.

Para isso teremos um arquivo bem simples, chamado `dicionario.txt`, cujo formato é uma palavra por linha:

```
alura
casa do codigo
caelum
```

```
desenvolvedor
programador
software
refatorar
code smell
```

Como ler os dados de um arquivo? Existem diversas maneiras de executar essa tarefa. Uma delas lê todo o conteúdo do arquivo de uma única vez para a memória:

```
texto = File.read("dicionario.txt")
```

Podemos depois quebrar o texto em um array de `Strings` separadas pelas quebras de linha, para isso separamos (*split*) pelas quebras de nova linha (*n*):

```
texto = File.read("dicionario.txt")
todas_as_palavras = texto.split("\n")
```

Agora que temos um array de palavras podemos escolher um número entre `0` e o total de palavras:

```
texto = File.read("dicionario.txt")
todas_as_palavras = texto.split("\n")
numero_aleatorio = rand(todas_as_palavras.size)
```

E escolher a palavra secreta:

```
texto = File.read("dicionario.txt")
todas_as_palavras = texto.split("\n")
numero_aleatorio = rand(todas_as_palavras.size)
palavra_secreta = todas_as_palavras[numero_aleatorio]
```

Pronto, basta colocarmos esse código em nossa função `sorteia_palavra_secreta`. Mas essa função está em `UI`, não em lógica:

```
def sorteia_palavra_secreta
    puts "Escolhendo uma palavra..."
    palavra_secreta = "programador"
```

```
    puts "Escolhida uma palavra com #{palavra_secreta.size}
          letras... boa sorte!"
    palavra_secreta
end
```

Vamos definir duas funções de `UI`, `avisa_escolhendo_palavra` e `avisa_palavra_escolhida`:

```
def avisa_escolhendo_palavra
    puts "Escolhendo uma palavra..."
end

def avisa_palavra_escolhida(palavra_secreta)
    puts "Escolhida uma palavra com #{palavra_secreta.size}
          letras... boa sorte!"
    palavra_secreta
end
```

Separamos uma função de lógica `sorteia_palavra_secreta`:

```
def sorteia_palavra_secreta
    avisa_escolhendo_palavra
    texto = File.read("dicionario.txt")
    todas_as_palavras = texto.split("\n")
    numero_aleatorio = rand(todas_as_palavras.size)
    palavra_secreta = todas_as_palavras[numero_aleatorio]
    avisa_palavra_escolhida palavra_secreta
end
```

14.2 Limpando a entrada de dados

Os dados que estamos lendo de nosso arquivo ainda podem possuir algumas sujeiras. Por exemplo, não queremos reclamar que o jogador errou caso ele chute a letra `P` maiúscula para a palavra `programador`, ou ainda que a palavra esteja registrada como `progrAmAdor` e que isso faça com que o chute `a` esteja errado.

Precisamos limpar, padronizar, normalizar, nossa entrada de dados. Podemos começar com nosso leitor de chutes, onde simplesmente transformamos o chute para letras minúsculas invocando o método `downcase`:

```
def pede_um_chute
    puts "Entre com a letra ou palavra"
    chute = gets.strip.downcase
    puts "Será que acertou? Você chutou #{chute}"
    chute
end
```

Ao ler nosso arquivo também devemos utilizar minúsculos para garantir que o chute, agora sempre em minúsculo, encontre sua posição correta nas palavras secretas. Para isso, ao escolher a palavra secreta, podemos transformá-la em minúsculo:

```
def sorteia_palavra_secreta
    avisa_escolhendo_palavra
    texto = File.read("dicionario.txt")
    todas_as_palavras = texto.split("\n")
    numero_aleatorio = rand(todas_as_palavras.size)
    palavra_secreta =
        todas_as_palavras[numero_aleatorio].downcase
    avisa_palavra_escolhida palavra_secreta
end
```

Pronto! Adicione no seu dicionário a palavra `RuBy` e teste-a com um `r` e `Y`, ambos devem funcionar. Você pode "roubar" e deixar temporariamente somente esta palavra no dicionário para forçar o teste a fazer sair a palavra que queria.

14.3 Processamento e memória devem ser otimizadas?

Por mais que essa abordagem funcione e seja totalmente válida, ler um arquivo inteiro para a memória pode ser ruim. Temporariamente consumimos muita memória e também toda vez que invocamos a função para sortear uma nova palavra temos que passar por todo o arquivo, mesmo que a palavra sorteada seja, por exemplo, a primeira. Imagine que se o arquivo possui diversos megas, ocuparemos todo esse espaço na memória para sortear uma única palavra. Outra opção seria ler primeiro um número no arquivo, indicando quantas palavras existem:

```
8
alura
casa do codigo
caelum
desenvolvedor
programador
software
refatorar
code smell
```

Após ler o número de linhas existentes:

```
arquivo = File.new("dicionario", "r")
total_de_palavras = arquivo.gets.to_i
```

Podemos sortear um número:

```
arquivo = File.new("dicionario", "r")
total_de_palavras = arquivo.gets.to_i
aleatoria = rand(total_de_palavras)
```

Agora vamos até a linha adequada, ignorando diversas delas:

```
arquivo = File.new("dicionario", "r")
total_de_palavras = arquivo.gets.to_i
aleatoria = rand(total_de_palavras)
for i = 1..aleatoria
    arquivo.gets
end
```

Finalmente lemos a palavra secreta, limpando-a, e fechamos o arquivo:

```
arquivo = File.new("dicionario", "r")
total_de_palavras = arquivo.gets.to_i
aleatoria = rand(total_de_palavras)
for i = 1..aleatoria
    arquivo.gets
end
palavra_secreta = arquivo.gets.strip.downcase
arquivo.close
```

[/box]

Cada abordagem de leitura de arquivo implica em um consumo de memória e processamento maior ou menor. Neste livro, nosso foco não é otimização, e um dicionário de 1 mega de palavras não faz cócegas aos computadores modernos, não precisamos nos preocupar com isso. Claro, no seu devido tempo, à medida que entramos em assuntos como otimização e análise de algoritmos essa preocupação aumenta, mas sempre questionando se ela faz sentido ou não. Aqui não há diferença prática no resultado.

Existem ainda outras formas de acesso a arquivo em disco, como o acesso a qualquer posição dele, sem precisar necessariamente ler as primeiras linhas (os primeiros bytes), mas para fazer isso você deve saber exatamente para que ponto do arquivo quer pular (em quantidade de bytes). Cada forma de acesso a um arquivo possui uma vantagem e desvantagem, como quase toda funcionalidade.

14.4 Escrita para arquivo: o melhor jogador

Devemos guardar quem foi o melhor jogador até agora, como forma de desafio ao próximo jogador. Primeiro vamos acumular os pontos por rodada. A função `joga` pode retornar os pontos até agora:

```
def joga(nome)
    # ...
    avisa_pontos pontos_ate_agora
    pontos_ate_agora
end
```

Ao invocarmos em nosso `jogo_da_forca` devemos acumular esses pontos, começando com zero:

```
def jogo_da_forca
    nome = da_boas_vindas
    pontos_totais = 0

    loop do
        pontos_totais += joga nome
        break if nao_quer_jogar?
```

```
    end
end
```

Precisamos avisar o jogador toda vez quais são seus pontos totais, logo definimos a função em `ui.rb`:

```
def avisa_pontos_totais(pontos)
    puts "Você possui #{pontos} pontos."
end
```

A cada nova rodada mostramos esses pontos para o jogador:

```
def jogo_da_forca
    nome = da_boas_vindas
    pontos_totais = 0

    loop do
        pontos_totais += joga nome
        avisa_pontos_totais pontos_totais
        break if nao_quer_jogar?
    end
end
```

Pronto, ao jogarmos nosso jogo e acertando as duas primeiras vezes temos o resultado de `200` pontos acumulados:

```
...
Será que acertou? Você chutou refatorar
Parabéns! Acertou!
Você ganhou 100 pontos.
Você possui 200 pontos.
Deseja jogar novamente? (S/N)
```

Antes de perguntarmos se o usuário deseja jogar novamente devemos salvar o nome do usuário e seus pontos em um arquivo, algo como:

```
def jogo_da_forca
    nome = da_boas_vindas
    pontos_totais = 0
```

```
    loop do
        pontos_totais += joga nome
        avisa_pontos_totais pontos_totais
        salva_rank nome, pontos_totais
        break if nao_quer_jogar?
    end
end
```

E definimos a função `salva_rank`:

```
def salva_rank(nome, pontos)
end
```

Definimos o conteúdo do arquivo, que será o nome na primeira linha e os pontos na segunda:

```
def salva_rank(nome, pontos)
    conteudo = "#{nome}\n#{pontos}"
end
```

Por fim, utilizamos a função `write` de `File` para salvar o conteúdo em um arquivo chamado `rank.txt`:

```
def salva_rank(nome, pontos)
    conteudo = "#{nome}\n#{pontos}"
    File.write "rank.txt", conteudo
end
```

Rodamos nosso jogo e após vencer duas vezes com `200` pontos temos a saída no arquivo:

```
guilherme
200
```

14.5 Lendo o melhor jogador

Queremos mostrar para o jogador atual quem é nosso campeão, e quantos pontos ele já conquistou. Já sabemos ler os dados de um arquivo e quebrar as linhas:

```
def le_rank
    conteudo_atual = File.read "rank.txt"
    dados = conteudo_atual.split("\n")
end
```

Agora que temos os dados do rank podemos mostrá-lo no começo do programa:

```
def jogo_da_forca
    nome = da_boas_vindas
    pontos_totais = 0

    avisa_campeao_atual le_rank

    loop do
        pontos_totais += joga nome
        avisa_pontos_totais pontos_totais
        salva_rank nome, pontos_totais
        break if nao_quer_jogar?
    end
end
```

E definir nossa função de `ui` que avisa o campeão atual:

```
def avisa_campeao_atual(dados)
    puts "Nosso campeão atual é #{dados[0]} com #{dados[1]}
          pontos."
end
```

Mas sempre salvamos o nome do último jogador e não é isso que queremos fazer. Desejamos sempre armazenar o melhor de todos os jogadores. Isso é, se o jogador já existe no arquivo, queremos manter somente o que possui mais pontos.

Antes de salvar, temos que verificar se o nome e a pontuação que já estão lá são mais baixas que a pontuação atual. Portanto, antes de salvar os dados devemos verificar sua pontuação:

```
def jogo_da_forca
    nome = da_boas_vindas
```

```
    pontos_totais = 0

    avisa_campeao_atual le_rank

    loop do
        pontos_totais += joga nome
        avisa_pontos_totais pontos_totais

        if le_rank[1].to_i < pontos_totais
            salva_rank nome, pontos_totais
        end

        break if nao_quer_jogar?
    end
end
```

Pronto! Já temos nosso rank de primeiro lugar. Sempre armazenamos quem é o melhor jogador.

14.6 Refatoração: extrair arquivo

Só falta organizar um pouco mais nosso código. Repare que temos duas funções de lógica ligadas ao rank. Na verdade mais do que lógica, elas são as funções de armazenamento de dados (*data access*), logo isolaremos tanto a `le_rank` quanto a `salva_rank` em um arquivo chamado `rank.rb`.

```
def le_rank
    conteudo_atual = File.read("rank.txt")
    dados = conteudo_atual.split("\n")
end

def salva_rank(nome, pontos)
    conteudo = "#{nome}\n#{pontos}"
    File.write("rank.txt", conteudo)
end
```

Não podemos esquecer de alterar nosso arquivo `forca.rb` para incluir relativo nosso `rank`:

```
require_relative 'ui'
require_relative 'rank'

# ...
```

Extrair código para outro arquivo não é uma refatoração grandiosa. Ela simplesmente varre para outro lugar o nosso código (que continua com funções globais), organizando em pequenas unidades (arquivos) para facilitar encontrar e manter as funções. Veremos em um próximo jogo como organizar ainda mais esse comportamento, fugindo da abordagem global.

14.7 A pilha de execução

Remova seu arquivo `rank.txt`. Agora rode novamente o jogo. Logo após entrar com seu nome, a aplicação para:

```
Começaremos o jogo para você, Guilherme
rank.rb:2:in `read': No such file or directory -
        rank.txt (Errno::ENOENT)
    from rank.rb:2:in `le_rank'
    from forca.rb:82:in `jogo_da_forca'
    from main.rb:3:in `<main>'
```

Como toda mensagem de erro, olhemos com calma o que acontece aqui. Primeiro, ele indica que o problema aconteceu no arquivo `rank.rb`, linha 2 (`rank.rb:2`). Ao tentarmos invocar a função `read` (`in 'read'`), ocorreu um erro com a descrição `No such file or directory - rank.txt (Errno: :ENOENT)`.

É fundamental entendermos a mensagem de erro para descobrir o que aconteceu. Ela diz que não foi encontrado o arquivo ou diretório `rank.txt`. Faz sentido, ele realmente não existe. Note que uma mensagem de erro realmente é rica de informações, ela nos em que arquivo, qual linha e qual a descrição que deu problema.

Mas o que são as informações que vêm logo depois dela? Temos mais três linhas para entender melhor:

```
from rank.rb:2:in `le_rank'
from forca.rb:82:in `jogo_da_forca'
from main.rb:3:in `<main>'
```

A próxima linha indica em que arquivo e linha estávamos quando o erro ocorreu:

```
conteudo_atual = File.read("rank.txt")
```

A próxima linha que ele indica é no arquivo de lógica:

```
avisa_campeao_atual le_rank
```

Por fim, a linha do arquivo `main.rb`:

```
jogo_da_forca
```

O que essas linhas indicam? Repare que a linha 3 do `main.rb` é quem chama a função `jogo_da_forca`, que é quem chama a função `le_rank`, que é quem chama `File.read`, que é onde acontece o erro. Isso é, essas linhas da mensagem de erro indicam o caminho que a execução do programa percorria quando do momento do erro. A única coisa "estranha" é que ela está de ponta-cabeça:

```
from rank.rb:2:in `le_rank'
from forca.rb:82:in `jogo_da_forca'
from main.rb:3:in `<main>'
```

Já simulamos nosso programa uma vez, vamos simulá-lo novamente. Agora com ainda mais carinho e atenção. Primeiro rodamos o comando `ruby main.rb`, que carrega o código do arquivo `main.rb`:

```
require_relative 'logic'

jogo_da_forca
```

A primeira linha carrega o arquivo de lógica, que por sua vez carrega os arquivos de `rank` e `ui`, todos definindo uma dezena de funções, mas nenhum executando algum código.

O programa chega até a terceira linha de nosso `main.rb`:

```
jogo_da_forca
```

Mas como o programa sabe em que linha ele está? Assim como uma criança precisa de seus dedos, uma variável, para saber em que número está, o programa também precisa de variáveis para indicar onde está no nosso código. Portanto, em algum lugar da memória, o programa armazena onde ele está:

```
funcao_atual = "main" # funcao sem nome, o código em si
arquivo_atual = "main.rb"
linha_atual = 3
```

Agora ele vai executar a linha 3. Mas ao tentar executar essa linha ele se depara com a invocação de uma função: ele precisa ir até onde está essa função, no arquivo `forca.rb`, linha `78`:

```
def jogo_da_forca
    nome = da_boas_vindas
    pontos_totais = 0

    avisa_campeao_atual le_rank

    loop do
        pontos_totais += joga nome
        avisa_pontos_totais pontos_totais

        if le_rank[1].to_i < pontos_totais
            salva_rank nome, pontos_totais
        end

        break if nao_quer_jogar?
    end
end
```

O programa troca o conteúdo das variáveis `arquivo_atual`, `linha_atual` e `funcao_atual`:

```
funcao_atual = "jogo_da_forca"
arquivo_atual = "forca.rb"
linha_atual = 78
```

A linha 78 é a definição da função, que não faz nada. Mas ele logo executa a linha 79, 80 e 81, chegando na 82 com a seguinte memória:

```
nome = "guilherme"
pontos_totais = 0

funcao_atual = "jogo_da_forca"
arquivo_atual = "forca.rb"
linha_atual = 82
```

Agora ele invoca a função `le_rank`, trocando o valor das variáveis novamente:

```
nome = "guilherme"
pontos_totais = 0

funcao_atual = "le_rank"
arquivo_atual = "rank.rb"
linha_atual = 1
```

Imagine que o código da função `le_rank` funcione normalmente, que o arquivo exista. Nesse caso, ao sair da função `le_rank`, para onde o programa deve ir? Para o resto da linha `82` no `forca.rb`, que é de onde o programa veio e invocou `le_rank`. Mas essa informação já era, já foi perdida quando trocamos o valor das variáveis para descrever `le_rank`. Opa. O programa ficaria perdido!

Isso significa que aquilo que descrevi não é a real descrição de como um programa funciona. Ele não pode armazenar somente a linha e arquivo atual. Se ele trocar o valor, fim. O programa não sabe para onde voltar e continuar quando uma função retorna.

Vamos escrever o problema com todas as palavras: o programa precisa saber exatamente quem chamou quem, e quem chamou quem, e quem chamou quem, ..., e quem chamou quem, para saber para quem ele deve voltar, um a um. Podemos pensar que cada vez que chamamos uma função, avançamos um nível, cada vez que saímos de uma função, saímos desse nível. Por exemplo, considere o código a seguir no arquivo `nomes.rb`:

```
def le_nome
    nome = gets # 1
    puts "Lido!" # 2
    nome
end
def pede_nome
    puts "Digite seu nome" # 3
    nome_lido = le_nome # 4
    puts "Pedido!" # 5
    nome_lido
end
def inicio
    nome = pede_nome # 6
    puts "Bem vindo #{nome}" # 7

    puts "Quero conhecer mais alguém"
    nome2 = pede_nome # 8
    puts "Olá #{nome2}" # 9
end

inicio # 10
```

O interpretador do Ruby lê a definição das funções `le_nome`, `pede_nome` e `inicio`. Por fim, ele chega à linha marcada `10`, onde antes de invocar a função `inicio` ele continua no código `principal` (*main*) de nosso arquivo. Portanto, temos só um nível de chamada:

```
nomes.rb:10 in main
```

Ao invocar a função `inicio`, ele vai para a linha marcada `6`, e anota o novo nível *em cima* do nível atual:

```
nomes.rb:6 in inicio
nomes.rb:10 in main
```

Essa informação nos diz exatamente como o programa chegou aonde está. Continuemos com a execução do código, invocando a função `pede_nome` e indo para a linha marcada `3`:

```
nomes.rb:3 in pede_nome
nomes.rb:6 in inicio
nomes.rb:10 in main
```

Ele agora imprime a mensagem `Digite seu nome` e chega à linha marcada 4.

```
nomes.rb:4 in pede_nome
nomes.rb:6 in inicio
nomes.rb:10 in main
```

```
nome_lido = le_nome # 4
```

Essa linha possui dois passos. O primeiro é a invocação da função `le_nome`, e o segundo a atribuição do resultado dela a uma variável local chamada `nome_lido`. Agora ele invoca a função `le_nome`:

```
nomes.rb:1 in le_nome
nomes.rb:4 in pede_nome
nomes.rb:6 in inicio
nomes.rb:10 in main
```

Note que agora leremos o nome do usuário, e chegamos à linha 2, portanto temos a existência de uma variável local:

```
def le_nome
    nome = gets # 1
    puts "Lido!" # 2
end
```

```
nomes.rb:2 in le_nome (nome="Guilherme")
nomes.rb:4 in pede_nome
nomes.rb:6 in inicio
nomes.rb:10 in main
```

Agora ele sai da função, como fazer isso? Simples, basta jogar fora a última função que foi invocada, a última linha que foi *empilhada* (*push*) nessa pilha de funções (*stack*):

```
nomes.rb:4 in pede_nome
nomes.rb:6 in inicio
nomes.rb:10 in main
```

Ainda existe o segundo passo a ser executado nessa linha. A função já retornou "Guilherme", logo é criada uma variável local chamada `nome_lido` com esse valor:

```
nomes.rb:4 in pede_nome (nome_lido = "Guilherme")
nomes.rb:6 in inicio
nomes.rb:10 in main
```

Chegamos à linha marcada `5`:

```
nomes.rb:5 in pede_nome (nome_lido = "Guilherme")
nomes.rb:6 in inicio
nomes.rb:10 in main
```

Note que durante sua execução, a variável antiga chamada `nome` já não existe mais, afinal, já saímos da função `le_nome`, mas a variável `nome_lido` local existe.

Terminando a execução da função `pede_nome`, o valor da variável `nome_lido` é retornado. Como fazemos mesmo a saída de uma função? Desempilhamos (*pop*) a última linha de nossa pilha de execução (*stack trace*).

```
nomes.rb:6 in inicio
nomes.rb:10 in main
```

A segunda parte da linha marcada `6` é a criação de uma variável chamada `nome`, com o valor do retorno da `pede_nome`:

```
nomes.rb:7 in inicio (nome = "Guilherme")
nomes.rb:10 in main
```

Imprimimos o `nome` com a mensagem de boas-vindas. Continuamos a execução até a linha marcada `8`:

```
def inicio
    nome = pede_nome # 6
    puts "Bem vindo #{nome}" # 7

    puts "Quero conhecer mais alguém"
    nome2 = pede_nome # 8
    puts "Olá #{nome2}" # 9
end
```

```
nomes.rb:8 in inicio (nome = "Guilherme")
nomes.rb:10 in main
```

Entramos na função `pede_nome`, que entra na função `le_nome`:

```
nomes.rb:1 in le_nome
nomes.rb:4 in pede_nome
nomes.rb:8 in inicio (nome = "Guilherme")
nomes.rb:10 in main
```

```
def le_nome
    nome = gets # 1
    puts "Lido!" # 2
end
```

O que acontecerá após executarmos a linha marcada `1`? Teremos criado uma nova variável local com valor, por exemplo, `Daniela`:

```
nomes.rb:2 in le_nome (nome = "Daniela")
nomes.rb:4 in pede_nome
nomes.rb:8 in inicio (nome = "Guilherme")
nomes.rb:10 in main
```

Calma aí. Duas variáveis com o mesmo nome? Isso faz sentido? Claro! Elas são locais à função! Cada uma delas só é visualizada dentro de seu próprio escopo, no caso local. A variável chamada `nome` que está sendo utilizada na função `inicio` é uma variável totalmente diferente da variável chamada `nome` que está sendo utilizada na função `le_nome`.

Este é o *local* do nome variável *local*. A variável vive e existe somente durante aquela chamada da função, e tem seu espaço de memória isolado da de outro método que foi chamado.

Assim como uma empilhadeira, a execução de um programa empilha as diversas funções que vão sendo invocadas. O processo de empilhar (*push*) e desempilhar (*pop*) é feito toda vez que entramos e saímos de uma função, permitindo que o programa saiba exatamente onde está e de onde veio. A pilha de coisas também tem nome, é a pilha de execução (*execution stack*). E aquelas linhas que mostram onde estamos, que nos mostram todo o caminho percorrido (*trace*) na pilha de execução, é o *stack trace*.

14.8 Resumindo

Aprendemos a fazer a leitura de um arquivo de textos, a limpar (normalizar) os dados lidos para evitar sujeira, e a escrever em um arquivo as informações do nosso melhor jogador. Refatoramos nosso código para extrair um arquivo com as responsabilidades de entrada e saída para disco e simulamos nosso programa para entender um conceito fundamental de toda linguagem de programação: a pilha de execução.

Capítulo 15

Arte ASCII: jogo da forca

15.1 Melhorando nossa interface com o usuário

Começamos criando nosso arquivo `rank.txt` novamente:

```
Guilherme
100
```

Podemos deixar o nosso jogo mais atraente mudando a maneira como nos comunicamos com nosso usuário final. Utilizando novamente *ASCII art* e um pouco de informação visual mais bonita vamos levar nosso jogo para um novo nível.

Mudamos a abertura para:

```
def da_boas_vindas
    puts "/****************/"
```

```
    puts "/ Jogo de Forca */"
    puts "/****************/"
    puts "Qual é o seu nome?"
    nome = gets.strip
    puts "\n\n\n\n\n\n"
    puts "Começaremos o jogo para você, #{nome}"
    nome
end
```

Na hora de desenhar a forca desejamos imprimir a cabeça, o corpo, as pernas e o braço:

```
def desenha_forca(erros)
    cabeca = "   "
    corpo = " "
    pernas = "   "
    bracos = "   "
    puts "  _______       "
    puts " |/      |      "
    puts " |      #{cabeca}  "
    puts " |      #{bracos}  "
    puts " |       #{corpo}     "
    puts " |      #{pernas}   "
    puts " |              "
    puts "_|___           "
    puts ""

end
```

No caso de o usuário ter errado pelo menos uma vez, desenhamos a cabeça:

```
if erros >= 1
    cabeca = "(_)"
end
```

Se ele errou pelo menos duas vezes, desenhamos um corpinho grande:

```
if erros >= 2
    bracos = " | "
```

```
    corpo = "|"
end
```

Com três erros desenhamos os braços:

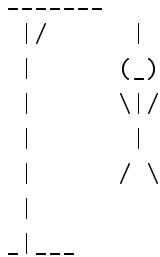
```
if erros >= 3
    bracos = "\\|/"
end
```

Por fim, com 4 erros desenhamos as pernas, ficando com a função `desenha_forca`:

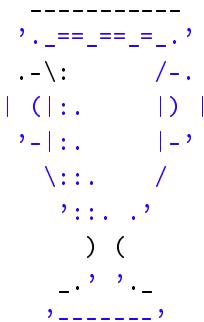
```
def desenha_forca(erros)
    cabeca = "   "
    corpo = " "
    pernas = "   "
    bracos = "   "
    if erros >= 1
        cabeca = "(_)"
    end
    if erros >= 2
        bracos = " | "
        corpo = "|"
    end
    if erros >= 3
        bracos = "\\|/"
    end
    if erros >= 4
        pernas = "/ \\"
    end

    puts "  -------       "
    puts " |/      |      "
    puts " |      #{cabeca}  "
    puts " |      #{bracos}  "
    puts " |       #{corpo}     "
    puts " |      #{pernas}   "
    puts " |              "
    puts "_|___           "
    puts ""
```

```
end
```

E a invocamos no cabeçalho:

```
def cabecalho_de_tentativa(chutes, erros, mascara)
    puts "\n\n\n\n"
    desenha_forca erros
    puts "Palavra secreta: #{mascara}"
    puts "Erros até agora: #{erros}"
    puts "Chutes até agora: #{chutes}"
end

def avisa_acertou_palavra
        puts "Parabéns, você ganhou!"

        puts "       ___________      "
        puts "      '._==_==_=_.'     "
        puts "      .-\\:      /-.    "
        puts "     | (|:.     |) |    "
        puts "      '-|:.     |-'     "
        puts "        \\::.    /      "
        puts "         '::. .'        "
        puts "           ) (          "
        puts "         _.' '._        "
        puts "        '-------'       "
end
```

Agora nosso jogo utiliza ASCII Art para permitir uma experiência mais agradável ao jogador:

```
_______
 |/     |
 |
 |
 |
 |
 |
_|___
```

```
Palavra secreta: _____
Erros até agora: 0
Chutes até agora: []

-------
 |/      |
 |      (_)
 |      \|/
 |       |
 |      / \
 |
_|___

Palavra secreta: a___a
Erros até agora: 4
Chutes até agora: ["a", "b", "c", "d", "e"]

Parabéns, você ganhou!

       -----------
      '._==_==_=_.'
      .-\:      /-.
     | (|:.     |) |
      '-|:.     |-'
        \::.    /
         '::. .'
           ) (
         _.' '._
        '-------'
```

Capítulo 16

Exercícios extras: jogo da forca

Aqui você encontra exercícios extras para soltar sua imaginação e praticar lógica de programação na linguagem Ruby.

16.1 Melhorando o jogo da forca

1) Ao perder, mostre a palavra secreta.

2) Simule seu jogo uma vez no papel.

3) Se ganhou, mostre o total de acertos.

4) Se você vira campeão e não era campeão até agora, mostre uma mensagem de parabéns.

5) Se você bateu seu próprio recorde, mostre uma outra mensagem de parabéns.

6) Não leia o arquivo de rank diversas vezes, armazene-o na memória em alguma variável.

7) Se o arquivo não existe, não mostre informação nenhuma sobre o rank.

8) O que acontece se o arquivo de dicionário tem uma linha em branco no fim? No meio? Como testar fácil? Faça com que não exista nenhum problema com arquivos com linhas em branco no começo, meio ou fim.

9) Se no arquivo de dicionário uma palavra possuir espaços antes ou depois, remova-os.

10) Somente letras ficam escondidas com o _. Todo o resto (números, hífen, pontuação etc.) são mostrados logo de cara para o jogador. Para isso você terá que verificar se o caractere é uma letra. Se for, confirma em relação aos chutes, se não é uma letra, já mostra.

11) Ao invés de toda vez recriar a `String` secreta com as letras não chutadas como _, tente manter o tempo todo uma `String` como _____________. À medida que o usuário chuta uma letra, troque as letras adequadas dessa `String` e use-a para imprimir e ajudar o usuário. Com isso nosso algoritmo passa a fazer apenas um laço por cada letra em vez do que ele faz hoje (dois laços).

12) Se você perder o jogo, mostre uma mensagem de fim de jogo:

```
def perdeu(palavra_secreta)
    puts
    puts "Puxa, você perdeu!"
    puts "A palavra era **#{palavra_secreta}**"
    puts

    puts "    ---------------         "
    puts "   /               \\       "
    puts "  /                 \\      "
    puts "//                   \\/\\  "
    puts "\\|   XXXX     XXXX   | /   "
    puts " |   XXXX     XXXX   |/     "
    puts " |   XXX       XXX   |      "
```

```
    puts " |                   |      "
    puts " \\__      XXX      __/     "
    puts "   |\\     XXX     /|       "
    puts "   | |           | |        "
    puts "   | I I I I I I I |        "
    puts "   |  I I I I I I  |        "
    puts "   \\_             _/       "
    puts "     \\_         _/         "
    puts "       \\_______/           "
end
```

16.2 Outros desafios

1) Considere que uma máquina de caixa eletrônico tem diversas notas de 100, 50, 20, 10, 5 e 1 real. Armazene cada quantidade inicial em uma variável (1 de 100, 2 de 50, 4 de 20, 8 de 10, 16 de 5 e 32 de 1). Peça para o usuário um valor, que é o quanto ele deseja sacar. Apresente para ele a solução que utiliza o menor número possível de notas para entregar-lhe o dinheiro de que precisa. Para isso tente sempre utilizar primeiro as notas maiores.

2) A cada saque, diminua do seu caixa os valores das notas que o usuário pediu.

3) Mude seu caixa eletrônico para pedir diversas vezes ao usuário valores a serem sacados.

4) Faça um programa que lê o arquivo `entrada.txt` e escreve para `saida.txt`. Ele deve compactar o arquivo de entrada da seguinte maneira: se uma letra (X) aparece uma única vez, escreva "1X", se ela aparece 5 vezes seguidas "5X". Se a letra é um número (por exemplo, uma vez o número 5), escreva "1-5".

5) Faça o programa que lê o arquivo de saída e devolve o arquivo descompactado.

Capítulo 17

Foge-foge, um jogo baseado no Pacman

17.1 Definindo a base do jogo e o mapa

Queremos desenvolver um jogo baseado no famoso Pacman, onde o herói foge de fantasmas que assombram sua vida. Nosso jogo, brasileiro, se chama foge-foge. Como de costume, começamos o jogo pedindo o nome de nosso usuário. Já sabemos que devemos isolar a parte de interface com o usuário, portanto criamos nosso `ui.rb`:

```
def da_boas_vindas
    puts "Bem-vindo ao Foge-foge"
    puts "Qual é o seu nome?"
    nome = gets.strip
```

```
    puts "\n\n\n\n\n\n"
    puts "Começaremos o jogo para você, #{nome}"
    nome
end
```

A lógica do jogo em `fogefoge.rb`:

```
require_relative 'ui'

def joga(nome)
    # nosso jogo aqui
end

def inicia_fogefoge
    nome = da_boas_vindas
    joga nome
end
```

E nosso `main.rb`, que inicia o jogo:

```
require_relative 'logic'

inicia_fogefoge
```

Precisamos definir nosso mapa. O jogo possui muros onde o herói não pode andar, fantasmas que caçam nosso jogador, o jogador em si e caminhos por onde ele pode passar trechos sem nada.

Chamando o muro de X, o mapa a seguir é um exemplo de quatro linhas e quatro colunas:

```
XXXX
X  X
X  X
XXXX
```

Se chamarmos o herói de H, o mapa a seguir é um mapa de quatro linhas e quatro colunas, com o herói no meio:

```
XXXX
XH X
```

```
X  X
XXXX
```

Já chamando de `F` um fantasma, temos o seguinte mapa difícil de ganhar válido:

```
XXXX
XH X
X FX
XXXX
```

Usando essas definições, o primeiro mapa válido que utilizaremos é com um único fantasma. Vamos criar o arquivo `mapa1.txt` a seguir. Não se esqueça de colocar os espaços em branco nas linhas cuja borda é um espaço em branco.

```
XXXXX
X H X
X X X
X X X
X   X
  X
 XXX
  X
X F X
XXXXX
```

Um mapa mais interessante com dois fantasmas, `mapa2.txt` seria:

```
XXXXXXXXX
X H     X
X X XXX X
X X X   X
X   X X X
  X     X
 XXX XX X
  X     X
X   X X X
XXXF  F X
XXX XXX X
```

```
XXX XXX X
XXX     X
```

Estamos usando letras para representar um mapa que nós, como seres humanos, conseguimos compreender. Poderíamos usar os e 1s, mas seria mais trabalhoso para entendermos o que estamos fazendo nesse instante, o que não é nosso foco.

17.2 Array de array: matriz

Baseados no segundo mapa, podemos escrever uma função de lógica (`logic.rb`) que lê tudo em uma `string` e quebra cada linha em um array de `strings`:

```
def le_mapa(numero)
    texto = File.read("mapa#{numero}.txt")
    mapa = texto.split("\n")
end
```

Mas que porcaria é um `"mapa#{numero}.txt"`. Sim, eu sou chato e gosto de dar nome a muitos bois. Nesse caso específico, se uma `string` tem um significado (uma semântica) outro que não seja meramente um texto, gosto de deixar isso bem claro. Parece ser um gosto bobo, mas deixar claro o que algo é facilita muito sua compreensão. Por exemplo, o que significa uma variável *pessoa*? E uma variável *usuarioLogado*? Ambos são pessoas, mas uma é claramente algo especial para meu programa, ela possui um significado para mim. Portanto, *extract variable* nele:

```
def le_mapa(numero)
    arquivo = "mapa#{numero}.txt"
    texto = File.read(arquivo)
    mapa = texto.split("\n")
end
```

Carregamos o mapa durante o jogo:

```
def joga(nome)
    mapa = le_mapa(1)
end
```

E o mostramos:

```
mapa = le_mapa(1)
desenha mapa
```

Claro, a cada nova rodada o usuário poderá andar. Para a direita, esquerda, cima ou baixo. Devemos sempre perguntar para onde ele quer ir, em nosso `ui.rb`:

```
def pede_movimento
    puts "Para onde deseja ir?"
    movimento = gets.strip
end
```

Durante o jogo fazemos um laço, mostrando o mapa e perguntando:

```
def joga(nome)
    mapa = le_mapa(1)
    while true
        desenha mapa
        direcao = pede_movimento
    end
end
```

Precisamos implementar a função `desenha` em nosso `ui.rb`, que é bem simples:

```
def desenha(mapa)
    puts mapa
end
```

Até aqui usamos somente recursos que já conhecíamos para ler o mapa e o movimento de nosso jogador. Testamos o jogo e a leitura está adequada:

```
Bem-vindo ao jogo do Pacman
...
Começaremos o jogo para você, Guilherme
XXXXX
X H X
X X X
```

```
X X X
X   X
  X
 XXX
  X
X F X
XXXXX
Para onde deseja ir?
```

Chamamos um array de arrays, um array de 2 dimensões, de uma matriz. Por mais que na matemática um array de uma dimensão (comumente chamado vetor) seja também chamado de matriz, no desenvolvimento de software costumamos utilizar o termo *array* para uma dimensão e *matriz* para duas ou mais dimensões. Mas tome cuidado: o que temos é um array de array ou um array de `String`?

17.3 Movimento

Nossa primeira funcionalidade do jogo envolve permitir ao jogador movimentar seu personagem. Queremos utilizar as mesmas teclas que jogos modernos utilizam: `WASD` para cima, esquerda, baixo e direita respectivamente.

Mas como encontrar o jogador no mapa? Vamos varrer o mapa e encontrá-lo:

```
def encontra_jogador(mapa)
    for linha = 0..(mapa.size-1)
        if mapa[linha].include? "H"
            # achei!
        end
    end
    # não achei!
end
```

Caso encontremos o caractere `H` dentro da linha do mapa, precisamos procurar em qual coluna, qual posição, ele está dentro:

```
def encontra_jogador(mapa)
    for linha = 0..(mapa.size-1)
```

```
        if mapa[linha].include? "H"
            for coluna = 0..(mapa[linha].size-1)
                if mapa[linha][coluna] == "H"
                    # achei!
                end
            end
        end
    end
    # não achei!
end
```

17.4 Refatorando

Já vi código feio, mas nosso código está horrendo. Cada linha faz duas, três, quatro ou até mesmo cinco coisas (`if`, `[]`, `[]`, `==`, definir o caractere mágico `"H"` na mesma linha).

Comecemos extraindo o `"H"`, que é o herói:

```
def encontra_jogador(mapa)
    caracter_do_heroi = "H"
    for linha = 0..(mapa.size-1)
        if mapa[linha].include? caracter_do_heroi
            for coluna = 0..(mapa[linha].size-1)
                if mapa[linha][coluna] == caracter_do_heroi
                    # achei!
                end
            end
        end
    end
    # não achei!
end
```

Agora podemos extrair a `linha_atual`:

```
def encontra_jogador(mapa)
    caracter_do_heroi = "H"
    for linha = 0..(mapa.size-1)
        linha_atual = mapa[linha]
        if linha_atual.include? caracter_do_heroi
```

```
            for coluna = 0..(linha_atual.size-1)
                if linha_atual[coluna] == caracter_do_heroi
                    # achei!
                end
            end
        end
    end
    # não achei!
end
```

Extraímos as condições:

```
def encontra_jogador(mapa)
    caracter_do_heroi = "H"
    for linha = 0..(mapa.size-1)
        linha_atual = mapa[linha]
        heroi_esta_nessa_linha =
            linha_atual.include? caracter_do_heroi
        if heroi_esta_nessa_linha
            for coluna = 0..(linha_atual.size-1)
                heroi_esta_aqui = linha_atual[coluna] ==
                    caracter_do_heroi
                if heroi_esta_aqui
                    # achei!
                end
            end
        end
    end
    # não achei!
end
```

Agora vamos parar para pensar um pouco mais. Como funciona o método `include?`? Ele passa caractere a caractere, verificando se o `"H"` está lá. Se é isso que ele faz, e refazemos isso dentro do `if`, não precisamos do `if`, afinal já fazemos o nosso próprio `for` para encontrar a posição adequada. A variável `heroi_esta_nessa_linha` vai embora:

```
def encontra_jogador(mapa)
    caracter_do_heroi = "H"
```

```
    for linha = 0..(mapa.size-1)
        linha_atual = mapa[linha]
        for coluna = 0..(linha_atual.size-1)
            heroi_esta_aqui = linha_atual[coluna] ==
                caracter_do_heroi
            if heroi_esta_aqui
                # achei!
            end
        end
    end
    # não achei!
end
```

17.5 O vazio, o nulo

Se já existe um método como o `include?` que diz se um caractere (ou `String`) está dentro de uma `String`, será que não existe um que já diz em qual posição ele aparece pela primeira vez? Procuramos na documentação e encontramos o método `index` que faz exatamente isso: devolve a posição onde o caractere está, ou nada (*nil*) caso não o encontre. Podemos usar essa função para encontrar nosso herói:

```
def encontra_jogador(mapa)
    caracter_do_heroi = "H"
    for linha = 0..(mapa.size-1)
        linha_atual = mapa[linha]
        coluna_do_heroi = linha_atual.index caracter_do_heroi
        if coluna_do_heroi
            # achei!
        end
    end
    # não achei!
end
```

Vazio, ou nulo, é definido como *nil* em Ruby. Não devemos nos confundir com uma *String* vazia:

```
string_vazia = ""
nada = nil
```

```
nulo = nil
vazio = nil
```

Podemos fazer um `if` para verificar o valor vazio tanto usando a comparação:

```
if coluna_do_heroi != nil
    # achei!
end
```

Quanto com um `if` simples. Em Ruby, tudo o que não é nulo nem falso é considerado verdadeiro:

```
if coluna_do_heroi
    # achei!
end
```

Temos que tomar cuidado quando temos um `nil` em nossas mãos. Suponha por exemplo que uma função retorne uma `String` ou `nil`. Nesse caso, se tentarmos calcular seu tamanho:

```
def pega_nome
  nil
end

nome = pega_nome
puts nome.size
```

Temos um erro que, por padrão, para toda a aplicação:

```
NoMethodError: undefined method `size' for nil:NilClass
```

Afinal, um valor vazio, nulo, não tem o método `size` como uma `String` tem. Tenha cuidado sempre que permitir a existência de um `nil` em seu código, e sempre verifique o retorno de uma função cuja documentação indica que pode retornar `nil`.

17.6 Laço funcional básico

Nosso código já está mais compreensível, mas ainda podemos melhorar. Será que não existe um laço que já nos traz tanto o contador (`linha`), quanto o valor daquele laço? Se não existisse, eu não perguntaria aqui? Talvez. Nesse caso, existe. Existe um método de `Array` (que `String` também tem) que permite passar por cada (*each*) elementos dele. Basta dizermos como queremos chamar essas variáveis. Cada elemento é nossa `linha_atual`:

```
def encontra_jogador(mapa)
    caracter_do_heroi = "H"
    mapa.each do |linha_atual|
        # cade a linha?
        coluna_do_heroi = linha_atual.index caracter_do_heroi
        if coluna_do_heroi != -1
            # achei!
        end
    end
    # não achei!
end
```

Mas qual o número da linha? Usamos o método `each_with_index` para nos dar a posição (*index*) de cada elemento, que chamaremos de `linha`:

```
def encontra_jogador(mapa)
    caracter_do_heroi = "H"
    mapa.each_with_index do |linha_atual, linha|
        coluna_do_heroi = linha_atual.index caracter_do_heroi
        if coluna_do_heroi != -1
            # achei!
        end
    end
    # não achei!
end
```

Essa maneira de programar, onde chamamos um método e passamos para ele um bloco de código (o conteúdo dentro do *do*) é o laço mais básico de uma maneira funcional de programar em Ruby.

17.7 Extraindo a posição

Agora que encontramos nosso herói, podemos retornar sua posição, tanto a linha quanto a coluna. Tome bastante cuidado pois estamos usando o formato linha/coluna, e não coluna/linha. Vamos colocara nosso código no `fogefoge.rb`:

```
def encontra_jogador(mapa)
    caracter_do_heroi = "H"
    mapa.each_with_index do |linha_atual, linha|
        coluna_do_heroi = linha_atual.index caracter_do_heroi
        if coluna_do_heroi != -1
            return [linha, coluna_do_heroi]
        end
    end
    # não achei!
end
```

O próximo passo é invocar a função em nosso laço principal:

```
desenha mapa
direcao = pede_movimento
heroi = encontra_jogador mapa
```

E movimentar de acordo. Para isso, se o usuário digitar `W`, devemos subir, diminuindo em `1` a linha atual:

```
desenha mapa
direcao = pede_movimento
heroi = encontra_jogador mapa
case direcao
    when "W"
        heroi[0] -= 1
end
```

Se ele digitar `S`, ele desce, aumentando em `1` a linha atual:

```
desenha mapa
direcao = pede_movimento
heroi = encontra_jogador mapa
```

```
case direcao
    when "W"
        heroi[0] -= 1
    when "S"
        heroi[0] += 1
end
```

Já no movimento horizontal, aumentamos um quando vamos para a direita e diminuímos um para a esquerda:

```
desenha mapa
direcao = pede_movimento
heroi = encontra_jogador mapa
case direcao
    when "W"
        heroi[0] -= 1
    when "S"
        heroi[0] += 1
    when "A"
        heroi[1] -= 1
    when "D"
        heroi[1] += 1
end
```

Falta agora reposicionar nosso jogador no mapa. Coloquemos o herói na nova posição:

```
desenha mapa
direcao = pede_movimento
heroi = encontra_jogador mapa
case direcao
    when "W"
        heroi[0] -= 1
    when "S"
        heroi[0] += 1
    when "A"
        heroi[1] -= 1
    when "D"
        heroi[1] += 1
```

```
end
mapa[heroi[0]][heroi[1]] = "H"
```

Antes de movimentá-lo, ele é tirado de sua posição, onde colocamos um espaço em branco. Ficamos com a função `joga`:

```
def joga(nome)
    mapa = le_mapa(1)
    while true
        desenha mapa
        direcao = pede_movimento
        heroi = encontra_jogador mapa
        mapa[heroi[0]][heroi[1]] = " "
        case direcao
            when "W"
                heroi[0] -= 1
            when "S"
                heroi[0] += 1
            when "A"
                heroi[1] -= 1
            when "D"
                heroi[1] += 1
        end
        mapa[heroi[0]][heroi[1]] = "H"
    end
end
```

Testamos o jogo e ele funciona!

```
XXXXX
X H X
X X X
X X X
X   X
  X
 XXX
  X
X F X
XXXXX
Para onde deseja ir?
```

```
D

XXXXX
X  HX
X X X
X X X
X   X
  X
 XXX
  X
X F X
XXXXX
Para onde deseja ir?
```

Lembre-se: estamos usando a letra maiúscula para mover-nos, não minúscula!

17.8 Refatorando

A complexidade da função `joga` aumentou rapidamente. Nosso `case` complica muito a compreensão, portanto extrairemos esse código. O que ele faz? Ele define a nova posição do herói? Vamos defini-lo:

```ruby
def calcula_nova_posicao(heroi, direcao)
    case direcao
        when "W"
            heroi[0] -= 1
        when "S"
            heroi[0] += 1
        when "A"
            heroi[1] -= 1
        when "D"
            heroi[1] += 1
    end
    heroi
end
```

E invocar a função:

```
def joga(nome)
    mapa = le_mapa(1)
    while true
        desenha mapa
        direcao = pede_movimento
        heroi = encontra_jogador mapa
        mapa[heroi[0]][heroi[1]] = " "
        nova_posicao = calcula_nova_posicao heroi, direcao
        mapa[nova_posicao[0]][nova_posicao[1]] = "H"
    end
end
```

17.9 Passagem por referência ou valor?

Para verificar a soma que efetuamos no array `heroi`, retornando seu novo valor, vamos imprimir as duas variáveis que temos:

```
def joga(nome)
    mapa = le_mapa(1)
    while true
        desenha mapa
        direcao = pede_movimento
        heroi = encontra_jogador mapa
        mapa[heroi[0]][heroi[1]] = " "
        nova_posicao = calcula_nova_posicao heroi, direcao
        puts "Antes: #{heroi}"
        puts "Depois: #{nova_posicao}"
        mapa[nova_posicao[0]][nova_posicao[1]] = "H"
    end
end
```

Rodamos o jogo e movemos para a direita:

```
...
D
Antes: [1, 3]
Depois: [1, 3]
...
```

Como assim? Os dois arrays têm o mesmo valor? O valor movido para a direita? O que está acontecendo aqui?

Vamos olhar com calma a simulação na memória de quando invocamos a função `calcula_nova_posicao`, afinal é ela quem devolve o array. O que acontece quando invocamos uma função com parâmetros? É criada uma nova variável, com o nome do parâmetro, e ela possui o mesmo valor que a variável que foi passada como argumento.

Isso significa que, ao passarmos `heroi` como argumento, pegamos todos os valores de `heroi` e copiamos em um novo `array`? Não. Se fizéssemos isso, imagine invocar uma função com um array de 1 mega como argumento. A cada vez que chamamos uma função, ele teria que copiar esse 1 mega de alguma maneira (existem diversas técnicas de otimização), mas o conteúdo mais cedo ou mais tarde seria copiado! Megas e mais megas! Nada bom.

Se toda vez que passássemos um array como argumento ele fosse copiado por inteiro, o tempo de processamento e o consumo de memória seria muito alto.

Ao mesmo tempo, o que é um array? É um "pedação" de memória em que cabe vários valores. Mas o que é uma variável que referencia um array? Quando temos uma variável que referencia um array, ela possui um único valor: um apontador para onde ele está. Isso é, toda variável possui um único valor. No caso de arrays, esse valor é um número que referencia o lugar na memória onde estão as casinhas para colocar diversos valores.

É justamente esse valor que é passado como parâmetro. Isso significa que, quando passamos um array como argumento, o que é copiado é o seu endereço, o apontador. O valor dele não é copiado, e tanto a variável anterior quanto a nova estão apontando para o mesmo afinal só existe um! array. Qualquer mudança que fizermos nesse array afeta as duas variáveis. E foi isso que aconteceu. Nossa variável `heroi` está referenciando o mesmo array que o `nova_posicao`.

Passamos uma referência para nosso `array` como argumento, não passamos uma cópia de nosso array. Esse foi nosso erro.

Para corrigi-lo, podemos pedir para o array ser duplicado (`dup`) assim que o recebemos:

```
def calcula_nova_posicao(heroi, direcao)
    heroi = heroi.dup
    case direcao
        when "W"
            heroi[0] -= 1
        when "S"
            heroi[0] += 1
        when "A"
            heroi[1] -= 1
        when "D"
            heroi[1] += 1
    end
    heroi
end
```

Agora sim:

```
...
D
Antes: [1, 2]
Depois: [1, 3]
...
```

Já podemos tirar os dois `puts`.

17.10 Detecção de colisão com o muro e o fim do mapa

Ainda existem algumas situações que não tratamos. A principal delas envolve bater com um muro. Não podemos permitir que nosso herói tente andar onde existe um muro. Para resolver isso, podemos verificar se sua nova posição possui um muro e, se sim, cancelar o movimento.

Agora que temos a posição exata à qual iremos nos movimentar, podemos conferir se ela é um muro e, se for, não andar:

```
def joga(nome)
    mapa = le_mapa(1)
    while true
```

```
        desenha mapa
        direcao = pede_movimento

        heroi = encontra_jogador mapa
        mapa[heroi[0]][heroi[1]] = " "
        nova_posicao = calcula_nova_posicao heroi, direcao
        if mapa[nova_posicao[0]][nova_posicao[1]] == "X"
            next
        end

        mapa[nova_posicao[0]][nova_posicao[1]] = "H"
    end
end
```

Mas cuidado! Só quero remover o herói de seu lugar caso ele tenha sido posicionado no lugar novo, ou seja, somente se o remarcarmos. Portanto, somente limpamos a posição no mapa caso ele não tenha batido no muro:

```
def joga(nome)
    mapa = le_mapa(1)
    while true
        desenha mapa
        direcao = pede_movimento

        heroi = encontra_jogador mapa
        nova_posicao = calcula_nova_posicao heroi, direcao
        if mapa[nova_posicao[0]][nova_posicao[1]] == "X"
            next
        end

        mapa[heroi[0]][heroi[1]] = " "
        mapa[nova_posicao[0]][nova_posicao[1]] = "H"
    end
end
```

Pronto! Nosso jogo já não permite mais movimentar para dentro de um muro. Falta verificarmos o fim do mapa. Como fazer isso?

O jogador não pode subir se estiver na linha `0`, nem ir para a esquerda se estiver na linha `0`. Isso é, não pode ir para a `nova_posicao[0] < 0`, nem

nova_posicao[1] < 0:

```
if nova_posicao[0] < 0
    next
end
if nova_posicao[1] < 0
    next
end
if mapa[nova_posicao[0]][nova_posicao[1]] == "X"
    next
end
```

Também não podemos deixar o jogador passar para "baixo" da última linha do mapa (mapa.size) nem para após a última coluna (mapa[0].size):

```
if nova_posicao[0] < 0
    next
end
if nova_posicao[1] < 0
    next
end
if nova_posicao[0] >= mapa.size
    next
end
if nova_posicao[1] >= mapa[0].size
    next
end
if mapa[nova_posicao[0]][nova_posicao[1]] == "X"
    next
end
```

Pronto, evitamos a colisão com muros e sair do mapa.

17.11 Refatorando com || e &&

Já temos a primeira parte de nosso jogo funcionando, afinal o jogador é capaz de andar pelo mapa. Mas vamos melhorar nosso código ainda mais antes

de continuar, uma vez que nossa função `joga` novamente ficou complexa demais.

Após calcular a nova posição, temos diversos `ifs`, que são equivalentes a um `switch`; são muitas condições, algo complicado de entender rapidamente. Vamos extrair uma função que diz se a posição é válida:

```
def posicao_valida?(mapa, nova_posicao)
    if nova_posicao[0] < 0
        return false
    end
    if nova_posicao[1] < 0
        return false
    end
    if nova_posicao[0] >= mapa.size
        return false
    end
    if nova_posicao[1] >= mapa[0].size
        return false
    end
    if mapa[nova_posicao[0]][nova_posicao[1]] == "X"
        return false
    end
    true
end
```

Mas seria ainda melhor se pudéssemos agrupar algumas dessas condições. Por exemplo, *se* a posição da linha for menor que `0`, ou maior ou igual ao número de linhas.

```
def posicao_valida?(mapa, nova_posicao)
    if nova_posicao[0] < 0 OU nova_posicao[0] >= mapa.size
        return false
    end
    if nova_posicao[1] < 0 OU nova_posicao[1] >= mapa[0].size
        return false
    end
    if mapa[nova_posicao[0]][nova_posicao[1]] == "X"
        return false
    end
```

```
    true
end
```

No Ruby é possível fazer a condição *OU* com o operador `||`:

```
def posicao_valida?(mapa, nova_posicao)
    if nova_posicao[0] < 0 OU nova_posicao[0] >= mapa.size
        return false
    end
    if nova_posicao[1] < 0 OU nova_posicao[1] >= mapa[0].size
        return false
    end
    if mapa[nova_posicao[0]][nova_posicao[1]] == "X"
        return false
    end
    true
end
```

> **|| E &&**
>
> Assim como o operador `||` (*OR*) retorna verdadeiro se a primeira ou segunda condição for verdadeira, o operador `&&` (*AND*) retorna verdadeiro somente se ambas forem verdadeiras.

```
def posicao_valida?(mapa, nova_posicao)
    if nova_posicao[0] < 0 || nova_posicao[0] >= mapa.size
        return false
    end
    if nova_posicao[1] < 0 || nova_posicao[1] >= mapa[0].size
        return false
    end
    if mapa[nova_posicao[0]][nova_posicao[1]] == "X"
        return false
    end
    true
end
```

Pronto, nossa função já está mais direta. Se extrairmos algumas variáveis o código fica bem mais legível:

```
def posicao_valida?(mapa, nova_posicao)
    linhas = mapa.size
    colunas = mapa[0].size

    estourou_linha =
        nova_posicao[0] < 0 || nova_posicao[0] >= linhas
    estourou_coluna =
        nova_posicao[1] < 0 || nova_posicao[1] >= colunas

    if estourou_linha || estourou_coluna
        return false
    end

    if mapa[nova_posicao[0]][nova_posicao[1]] == "X"
        return false
    end

    true
end
```

Por fim, renomeamos `nova_posicao` para `posicao`:

```
def posicao_valida?(mapa, posicao)
    linhas = mapa.size
    colunas = mapa[0].size

    estourou_linha = posicao[0] < 0 || posicao[0] >= linhas
    estourou_coluna = posicao[1] < 0 || posicao[1] >= colunas

    if estourou_linha || estourou_coluna
        return false
    end

    if mapa[nova_posicao[0]][nova_posicao[1]] == "X"
        return false
    end

    true
end
```

Nossa função joga invoca a posicao_valida?:

```
def joga(nome)
    mapa = le_mapa(1)
    while true
        desenha mapa
        direcao = pede_movimento

        heroi = encontra_jogador mapa
        nova_posicao = calcula_nova_posicao heroi, direcao
        if !posicao_valida? mapa, nova_posicao
            next
        end

        mapa[heroi[0]][heroi[1]] = " "
        mapa[nova_posicao[0]][nova_posicao[1]] = "H"
    end
end
```

17.12 Duck typing na prática

Até agora usamos um array de Strings para representar nosso mapa. Poderíamos usar um array de arrays. Como no nosso caso utilizamos letras, não parece existir nenhuma vantagem em utilizar números (ou caracteres soltos em um array). Mas também não há nenhum método de String que estamos utilizando. Tudo indica que a única coisa que utilizaremos é extrair o valor na posição de uma String, com o [], e o tamanho de uma linha com o size. Como tanto array quanto String respondem a esses métodos, tanto faz por enquanto nossa abordagem.

Tanto faz se é um Array ou String, só me preocupo se ele responde ao comportamento de que preciso. No mundo animal existe uma analogia famosa (e um tanto estranha) com patos: se ele faz *quack* como um pato, não me importa se é um pato, ele faz *quack* como um pato.

Por um lado, o artifício de invocar um comportamento, independente do tipo que estamos utilizando (chamado de *duck typigin*), é poderoso, mas pode trazer problemas. O método size de um Array pode trazer o número de

elementos dele, o de `String` também. Portanto, qualquer coisa que tem `size` me satisfaz. Será?

Mas e se eu tenho um tipo chamado `Product` (*Produto*), cujo tamanho físico o método `size` devolve? `size` em um contexto tem um significado, em outro tem outro. Boa sorte.

A frase original do *duck typing* era *When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck.*. Em tradução livre, "Quando vejo um pássaro que anda como um pato, nada como um pato e grasna como um pato, eu chamo esse pássaro de pato" (James Whitcomb Riley). O problema é que muito desenvolvedor (inclusive o wikipedia) interpreta isso como uma abstração total da tipagem, em uma frase resumida a "Se anda como um pato, e grasna como um pato, é um pato".

Repare a diferença entre a original e a resumida. Na frase original ainda nos preocupamos com a tipagem daquilo com que estamos trabalhando: ainda tem que ser um ser vivo, mais ainda, um animal, mais ainda, um pássaro. Se um homem grasna, anda e nada como um pato, ele não é um pato. E é ai que moram o risco e a vantagem do *duck typing* em Ruby: ele não verifica nada da tipagem, ele trabalha com a versão reduzida da frase, e não a original. Filosoficamente, não existe "verdade" ou "mentira" na frase original ou resumida. Existem consequências positivas e negativas de toda funcionalidade de uma linguagem, é importante sempre aprendermos todas elas.

Linguagens com *duck typing* nos alegram ao permitir tais invocações e ao mesmo tempo pecam em não nos protegerem de erros como esses. Não só um programa com um `Product` rodaria, como o seu resultado seria totalmente inesperado boa sorte.

Tome cuidado com a ilusão que o *duck typing* às vezes nos passa. Não seja enganado quando ensinarem somente o lado positivo de uma funcionalidade de uma linguagem. Sempre aprendemos o bom e o ruim para tomar nossas decisões conscientemente. Esse é o foco desse material. Neste caso atual, você não quer saber o *tamanho* de qualquer coisa, você quer saber o tamanho de seu mapa, e isso implica em um significado (uma semântica) bem específica, que a linguagem Ruby não fornece. Por outro lado, ela permite trocar um tipo por outro "sem nos preocupar" muito.

17.13 FOR I X FOR LINHA

Uma prática muito comum no mundo de programação é a abreviação e padronização do nome de variáveis. São diversos os `tots` que são totais, ou os `s_nome` que são `string` nome. O padrão usado por desenvolvedores Ruby é o de não utilizar um caractere no começo de uma variável para indicar qual o tipo daquela variável.

Por outro lado, existem nomes de métodos e variáveis abreviados e outros não. Como boa prática e regra geral, não abrevie. Quanto mais curto, mais ambíguo e maior a chance de conflito. Como citado anteriormente, `size` é ambíguo, mas `tamanho` e `quantidade_de_elementos` não. Tudo depende do que você precisa, mas como regra geral evite a ambiguidade e o possível erro do desenvolvedor justamente por não entender o que está fazendo.

Um exemplo clássico de abreviação é o uso da variável `i` para um laço:

```
for i = 0..(mapa.size-1)
    puts mapa[i]
end
```

Se o código dentro de seu laço utilizar a variável que criou para algo importante, não deixe seu nome como um mero `i`, defina-a com o valor real que ela tem, com seu significado dentro do contexto atual:

```
for linha = 0..(mapa.size-1)
    puts mapa[linha]
end
```

Não foi sem querer que até agora não vimos nenhum `for i`. Em todos os instantes que fizemos um laço, as variáveis possuíam algum significado real para nossa aplicação, e ao darmos um nome real, deixamos claro para o próximo desenvolvedor o que aquela variável representa.

17.14 RESUMINDO

A definição de um formato de entrada e saída é um passo fundamental para todo programa que vai gravar dados em algum lugar. Esses arquivos de texto

são simples mas já implicam em entendermos como funciona o processo de *input* e *output* (*IO*).

Vimos como definir arrays de arrays, apesar de usarmos até agora um array de `Strings`. Fomos capazes de movimentar nosso jogador, entender o que significa o vazio, o nulo (*nil*), utilizar um laço funcional básico (*each*).

Outra base da programação foi apresentada aqui: a passagem de parâmetros por valor e por referência. Refatoramos nosso código para usar os operadores chamados de *short-circuit*, :`&&` e `||`. Por fim, vimos quais são as implicações positivas até agora e o cuidado que devemos tomar com *duck typing*.

Capítulo 18

Botando os fantasmas para correr: arrays associativos, duck typing e outros

18.1 Array associativo: case e +1, -1

Já falamos anteriormente que sequências de `ifs`, e o uso de `cases` são possíveis *code smells*. Eles indicam que pode estar presente algo sujo, que pode atrapalhar a manutenção do código ou facilitar o aparecimento de novos bugs.

Nossa função `calcula_nova_posicao` demonstra exatamente este cenário:

```
def calcula_nova_posicao(heroi, direcao)
    heroi = heroi.dup
```

```
    case direcao
        when "W"
            heroi[0] -= 1
        when "S"
            heroi[0] += 1
        when "A"
            heroi[1] -= 1
        when "D"
            heroi[1] += 1
    end
    heroi
end
```

Como mudar nosso código para evitar esse `case`? Repare que o que fazemos aqui é basicamente mapear uma mudança de posição para cada tecla, e a mudança é feita tanto na linha (posição `0` do array) quanto na coluna (posição `1`).

```
def calcula_nova_posicao(heroi, direcao)
    heroi = heroi.dup
    case direcao
        when "W"
            heroi[0] -= 1
            heroi[1] += 0
        when "S"
            heroi[0] += 1
            heroi[1] += 0
        when "A"
            heroi[0] += 0
            heroi[1] -= 1
        when "D"
            heroi[0] += 0
            heroi[1] += 1
    end
    heroi
end
```

Podemos padronizar mais ainda, utilizando somente somas, seja de um número positivo ou negativo:

```
def calcula_nova_posicao(heroi, direcao)
    heroi = heroi.dup
    case direcao
        when "W"
            heroi[0] += -1
            heroi[1] += 0
        when "S"
            heroi[0] += 1
            heroi[1] += 0
        when "A"
            heroi[0] += 0
            heroi[1] += -1
        when "D"
            heroi[0] += 0
            heroi[1] += 1
    end
    heroi
end
```

Podemos também padronizar criando duas variáveis, `anda_linha` e `anda_coluna`:

```
def calcula_nova_posicao(heroi, direcao)
    heroi = heroi.dup
    case direcao
        when "W"
            anda_linha += -1
            anda_coluna += 0
        when "S"
            anda_linha += +1
            anda_coluna += 0
        when "A"
            anda_linha += 0
            anda_coluna += -1
        when "D"
            anda_linha += 0
            anda_coluna += +1
    end
    heroi[0] += anda_linha
```

```
    heroi[1] += anda_coluna
    heroi
end
```

Mas Guilherme, seu código está ainda maior que o que possuíamos antes. É verdade, ele está maior, mas mais claro. Ele diz que, se você me passa determinado caractere, eu faço você andar um número determinado de linhas e colunas, isso é, eu *mapeio* uma letra (uma chave), para um número (um valor).

W anda -1 e 0. S anda +1 e 0. A anda 0 e -1. D anda 0 e +1. Eu crio uma conexão entre uma chave (nesse caso, o caractere) e um valor (o quanto ele anda em cada dimensão, linha e coluna). Adivinha? Ruby já tem essa estrutura, alguém que mapeia algo para outra coisa, assim como um dicionário mapeia palavras para suas definições. Aqui um dicionário mapeará, conectará um caractere com o quanto devemos andar:

```
heroi = heroi.dup
movimentos = {
    "W" => [-1, 0],
    "S" => [+1, 0],
    "A" => [0, -1],
    "D" => [0, +1]
    }
```

Usamos o dicionário como se fosse um array, afinal ele é um array, mas um array que associa chave e valor, um *array associativo*:

```
heroi = heroi.dup
movimentos = {
    "W" => [-1, 0],
    "S" => [+1, 0],
    "A" => [0, -1],
    "D" => [0, +1]
    }
movimento = movimentos[direcao]
heroi[0] += movimento[0]
heroi[1] += movimento[1]
```

Isso é, trocamos nosso `case` ou sequência de `ifs` por um array associativo:

```
def calcula_nova_posicao(heroi, direcao)
    heroi = heroi.dup
    movimentos = {
        "W" => [-1, 0],
        "S" => [+1, 0],
        "A" => [0, -1],
        "D" => [0, +1]
        }
    movimento = movimentos[direcao]
    heroi[0] += movimento[0]
    heroi[1] += movimento[1]
    heroi
end
```

Existe uma boa e válida discussão aqui, se essa refatoração em si é justificável. O código final é mais explícito, mas utilizamos um artifício para chegar aonde queríamos. Nesse código, por exemplo, o que acontece se utilizarmos uma tecla inválida? E no código anterior, o que aconteceria?

Claro, tratamento de tecla inválida deve ser feito de qualquer maneira, mas a utilização de dicionários (arrays associativos, mapas etc.) como instrumento para execução de lógica de negócios tem que ser tratada com cuidado. Pense sempre se o código final justificou sua utilização.

Acredito que o código no nosso exemplo atual é mais explícito no que está fazendo e quanto ao resultado final da função. Mas não acredito que arrays associativos possam ser abusados como recurso lógico. Como regra geral, verifique a quantidade extra de `ifs` que vai adicionar ao seu código somente por usar um array associativo. No final das contas, valeu a pena?

18.2 Movimento dos fantasmas: o desafio no duck typing

Chegou a hora de nossos inimigos se moverem. Começaremos alterando nosso código para que o mapa carregado seja o segundo:

```
def joga(nome)
    mapa = le_mapa(2)
    # ...
end
```

No fim de nossa jogada, movemos os fantasmas:

```
def joga(nome)
    mapa = le_mapa(2)
    while true
        # ...

        move_fantasmas mapa
    end
end
```

Para o primeiro passo de "inteligência" de nosso inimigo, faremos com que os fantasmas andem sempre para a direita. Como fazer isso? Queremos pegar todos os fantasmas e andar uma casa para a direita. Quebremos essa afirmação, como crianças, em passos básicos para cada uma das frases:

1) Queremos pegar todos os fantasmas;

2) e andar uma casa para a direita.

Ótimo, primeiro queremos todos os fantasmas. Como encontrá-los? Podemos varrer nosso array de strings, procurando o caractere `F`. Para isso, primeiro passamos por todas as linhas:

```
def move_fantasmas(mapa)
    caracter_do_fantasma = "F"
    mapa.each_with_index do |linha_atual, linha|
    end
end
```

Agora passamos por cada caractere de uma linha:

```
def move_fantasmas(mapa)
    caracter_do_fantasma = "F"
```

```
    mapa.each_with_index do |linha_atual, linha|
        linha_atual.each_with_index do |caractere_atual, coluna|
        end
    end
end
```

Verificamos se ele é um fantasma e, se sim, movemos:

```
def move_fantasmas(mapa)
    caracter_do_fantasma = "F"
    mapa.each_with_index do |linha_atual, linha|
        linha_atual.each_with_index do |caractere_atual, coluna|
            eh_fantasma = caractere_atual == caracter_do_fantasma
            if eh_fantasma
                move_fantasma
            end
        end
    end
end
```

Agora que encontramos todos eles, precisamos implementar o `move_fantasma` com o qual o fazemos andar. Primeiro o tiramos da posição atual:

```
def move_fantasma(mapa, linha, coluna)
    mapa[linha][coluna] = " "
end
```

Movemos para direita:

```
def move_fantasma(mapa, linha, coluna)
    mapa[linha][coluna] = " "
    linha += 0
    coluna += 1
end
```

E o colocamos na nova posição:

```
def move_fantasma(mapa, linha, coluna)
    mapa[linha][coluna] = " "
```

```
    linha += 0
    coluna += 1
    mapa[linha][coluna] = "F"
end
```

Ao invocar, passamos os parâmetros necessários:

```
move_fantasma mapa, linha, coluna
```

Pronto. Nossos fantasmas podem andar. Vamos testar uma rodada:

```
fogefoge.rb:31:in `block in move_fantasmas':
    undefined method `each_with_index' for
        "XXXXXXXXX":String (NoMethodError)
```

O que acontece? `String` não possui o método `each_with_index`? Nem tudo que parece um pato é um pato. Antes eu queria alguém com `each`, `size` e `[]`. Agora eu quero com `each_with_index`. Eu enganei a mim mesmo, menti para mim quando disse que queria alguém que se parecesse com um array, que uma `String` bastava. Eu já estava atrelado à existência e significado de três métodos de um array, agora estou a quatro. Não parece ser pouca coisa o suficiente para dizer que eu queria somente alguem que se comportasse como um array. Eu queria um array, estou triste.

Deveríamos ter trocado antes? Agora? Nunca deveríamos ter trabalhado com `String`? A discussão é longa e com ótimos argumentos para os dois lados. Para nosso aprendizado esse é o momento ideal para efetuarmos a troca, e visualizarmos a vantagem e uma primeira desvantagem do uso de *duck typing*. A segunda desvantagem é quando o código roda, e o efeito é inesperado. A grande vantagem era até agora não ter precisado se preocupar com isso.

O que fazer para corrigir nosso problema? Uma solução seria ao ler o mapa, transformar as linhas de `Strings` em `Arrays` de caractere, utilizando o método `chars` em cada linha. Mas a impressão de um array de caracteres ficaria horrível, teríamos que reconcatenar os caracteres para formar uma `String`. Note que realmente quem parece um pato não é um pato. *Todo* momento que usamos nosso mapa, sabemos que ele era um array de `String`. Mudar o tipo é mudar tudo, não é mudar pouco.

Nossa solução será, portanto, simples. Ao precisarmos do método `each_with_index` invocaremos o método `chars`:

```
def move_fantasmas(mapa)
    caracter_do_fantasma = "F"
    mapa.each_with_index do |linha_atual, linha|
        linha_atual.chars.each_with_index do |caractere_atual,
        coluna|
            eh_fantasma = caractere_atual == caracter_do_fantasma
            if eh_fantasma
                move_fantasma mapa, linha, coluna
            end
        end
    end
end
```

Agora sim, rodamos o jogo, movemos uma para a esquerda e temos os fantasmas andando uma casa para a direita:

```
...
XXXXXXXXX
X H     X
X X XXX X
X X X   X
X   X X X
  X     X
 XXX XX X
  X     X
X   X X X
XXXF  F X
XXX XXX X
XXX XXX X
XXX     XPara onde deseja ir?
A
XXXXXXXXX
XH      X
X X XXX X
X X X   X
X   X X X
  X     X
 XXX XX X
  X     X
```

```
X   X X X
XXX F  FX
XXX XXX X
XXX XXX X
XXX     X
Para onde deseja ir?
```

A sacada do *duck typing* é que ele é genial ao permitir que o interpretador não se preocupe com a tipagem nem em tempo de compilação (na implemetação padrão do Ruby, compilação é quando ele valida a sintaxe do seu arquivo), nem durante a execução do código. O interpretador somente está preocupado se o valor referenciado pela nossa variável responde à função, ao método, em questão. Se sim, ótimo.

Por outro lado, nós como desenvolvedores estamos preocupados com a tipagem ao escrevermos nosso código. Não importa ter um valor que responda ao método `size` se daqui a pouco preciso do método `each_with_index`, o que eu quero agora é alguém que se comporte quase como um `array`, alguém que tenha os mais e mais métodos de `array`. É arriscado afirmar que estamos totalmente atrelados ao tipo em si, mas uma vez que estamos subordinados ao significado do método que invocamos, estamos atrelados a algo além do que somente sua existência, porém o interpretador não está preocupado com o tipo. Assim podemos definir a dinâmica de tipos no Ruby: o interpretador não se preocupa com eles, nós nos preocupamos sempre que precisamos escrever ou alterar nosso código.

18.3 Movimento dos fantasmas: reutilização de função

Mas se andarmos novamente, agora para baixo, os fantasmas movem mais um para a direita, entrando em contato com o muro.

```
...
XXXXXXXXX
X       X
XHX XXX X
X X X   X
```

```
X   X X X
  X     X
 XXX XX X
  X     X
X   X X X
XXX F  FX
XXX XXX X
XXX XXX X
XXX     X
Para onde deseja ir?
S
XXXXXXXXX
X       X
X X XXX X
XHX X   X
X   X X X
  X     X
 XXX XX X
  X     X
X   X X X
XXX  F  F
XXX XXX X
XXX XXX X
XXX     X
```

Faltou validarmos a posição de nossos fantasmas. Já temos a função `posicao_valida?`, basta invocá-la verificando se o resultado é válido antes de mover o fantasma. Começamos criando a nova posição em um array:

```
def move_fantasma(mapa, linha, coluna)
    posicao = [linha, coluna + 1]
end
```

Só alteramos os valores de nosso mapa se a posição for válida:

```
def move_fantasma(mapa, linha, coluna)
    posicao = [linha, coluna + 1]
    if posicao_valida? mapa, posicao
        mapa[linha][coluna] = " "
        mapa[posicao[0]][posicao[1]] = "F"
```

```
    end
end
```

Caso a posição não seja válida, o fantasma fica parado. Testamos nos movimentar duas vezes e agora o fantasma fica encostado na parede, não passa por ela:

```
XXXXXXXXX
X  H    X
X X XXX X
X X X   X
X   X X X
  X     X
 XXX XX X
  X     X
X   X X X
XXX F  FX
XXX XXX X
XXX XXX X
XXX     X
Para onde deseja ir?
D
XXXXXXXXX
X   H   X
X X XXX X
X X X   X
X   X X X
  X     X
 XXX XX X
  X     X
X   X X X
XXX  F FX
XXX XXX X
XXX XXX X
XXX     X
```

18.4 Fantasma contra fantasma?

Se andarmos um quarto passo, o mapa se torna:

```
...
XXXXXXXXX
X       X
X X XXX X
X X X   X
XH  X X X
  X     X
 XXX XX X
  X     X
X   X X X
XXX    FX
XXX XXX X
XXX XXX X
XXX     X
```

O que aconteceu? Um dos fantasmas sumiu. Acontece que, enquanto um dos fantasmas ficou parado, o outro tomou seu lugar. Com isso, perdemos um deles. Devemos proibir um fantasma de andar para a posição de outro fantasma. A solução inicial é não permitir que ele ande para uma casa onde já existe um `F` marcado. Aliás, não faz sentido o jogador ir em direção ao fantasma. Portanto alteramos nossa função de `posicao_valida?` para conferir se ela possui muro (`X`) ou fantasma (`F`):

```
if mapa[posicao[0]][posicao[1]] == "X"
    return false
end
if mapa[posicao[0]][posicao[1]] == "F"
    return false
end
```

Opa, se estamos usando essa posição duas vezes, *extract variable* fica mais claro:

```
valor_local = mapa[posicao[0]][posicao[1]]
if valor_local == "X" || valor_local == "F"
    return false
end
```

Agora sim, após andar diversos passos para a direita os fantasmas não brigam mais, ficam encostados um ao outro:

```
XXXXXXXXX
X      HX
X X XXX X
X X X   X
X   X X X
  X     X
 XXX XX X
  X     X
X   X X X
XXX   FFX
XXX XXX X
XXX XXX X
XXX     X
```

18.5 Resumindo

Vimos como funciona o dicionário, um array associativo, e tivemos que tomar cuidado com mudanças de +1 e -1, algo extremamente perigoso em programação. Entendemos melhor o que significa o *duck typing* e sua vantagem além de corrigir um bug que surgiu no jogo, onde dois fantasmas ocupavam a mesma posição. Vimos que o duck typing permite que o interpretador fique despreocupado em relação à tipagem, mas que nós ainda nos preocupamos com o significado e existência dos métodos, portanto com quem os definiu, seus tipos. Vimos também que tanto bugs quanto funcionalidades são descritos através de estruturas de lógica, com controle de fluxo e laços.

Capítulo 19

Matrizes e memória

19.1 Teletransportando fantasmas: cuidados a tomar com a memória

Ainda falta um pouco mais de inteligência em nossos fantasmas. Eles têm que andar para qualquer uma das quatro direções, aleatoriamente, e sempre lembrando que só podem andar em espaços válidos.

Como implementar isso? Teremos que mudar nosso `move_fantasma` para escolher uma nova `posicao` válida. Nosso código atual simplesmente move o fantasma para a direita:

```
def move_fantasma(mapa, linha, coluna)
    posicao = [linha, coluna + 1]
    if posicao_valida? mapa, posicao
        mapa[linha][coluna] = " "
```

```
        mapa[posicao[0]][posicao[1]] = "F"
    end
end
```

Primeiro calculamos um lado:

```
def posicoes_validas_a_partir_de(mapa, posicao)
    posicoes = []
    baixo = mapa[posicao[0] + 1][posicao[1]]
end
```

Agora verificamos se ela é válida, adicionando ao nosso array:

```
def posicoes_validas_a_partir_de(mapa, posicao)
    posicoes = []
    baixo = mapa[posicao[0] + 1][posicao[1]]
    if posicao_valida? baixo
        posicoes << baixo
    end
end
```

Fazemos a mesma coisa para a direita:

```
def posicoes_validas_a_partir_de(mapa, posicao)
    posicoes = []
    baixo = mapa[posicao[0] + 1][posicao[1]]
    if posicao_valida? baixo
        posicoes << baixo
    end
    direita = mapa[posicao[0]][posicao[1] + 1]
    if posicao_valida? direita
        posicoes << direita
    end
end
```

Para cima e para baixo:

```
def posicoes_validas_a_partir_de(mapa, posicao)
    posicoes = []
    # ...
```

```
    cima = mapa[posicao[0] - 1][posicao[1]]
    if posicao_valida? cima
        posicoes << cima
    end
    esquerda = mapa[posicao[0]][posicao[1] - 1]
    if posicao_valida? esquerda
        posicoes << esquerda
    end
end
```

Por fim, retornamos nosso array:

```
def posicoes_validas_a_partir_de(mapa, posicao)
    posicoes = []
    # ...
    posicoes
end
```

Já temos nossas posições válidas para o fantasma. Agora precisamos utilizá-las:

```
def move_fantasma(mapa, linha, coluna)
    posicoes = posicoes_validas_a_partir_de mapa, [linha, coluna]
    posicao = # escolhe uma posicao

    mapa[linha][coluna] = " "
    mapa[posicao[0]][posicao[1]] = "F"
end
```

Faremos o fantasma seguir a primeira posição possível:

```
def move_fantasma(mapa, linha, coluna)
    posicoes = posicoes_validas_a_partir_de mapa, [linha, coluna]
    posicao = posicoes[0]

    mapa[linha][coluna] = " "
    mapa[posicao[0]][posicao[1]] = "F"
end
```

Mas precisamos verificar que existe alguma posição para a qual mover, claro. Perguntamos ao nosso array se ele está vazio:

```
def move_fantasma(mapa, linha, coluna)
    posicoes = posicoes_validas_a_partir_de mapa, [linha, coluna]
    if posicoes.empty?
        return
    end

    posicao = posicoes[0]

    mapa[linha][coluna] = " "
    mapa[posicao[0]][posicao[1]] = "F"
end
```

Testamos agora e os fantasmas andam, um para baixo, um para a direita:

```
XXXXXXXXX
X H     X
X X XXX X
X X X   X
X   X X X
  X     X
 XXX XX X
  X     X
X   X X X
XXXF  F X
XXX XXX X
XXX XXX X
XXX     X
Para onde deseja ir?
D
XXXXXXXXX
X  H    X
X X XXX X
X X X   X
X   X X X
  X     X
 XXX XX X
  X     X
X   X X X
XXX    FX
XXX XXX X
```

```
XXX XXX X
XXX F   X
```

19.2 Corrigindo o teletransporte

Mas aconteceu muito mais do que isso. O fantasma da esquerda se teletransportou? Teletransporte vale? Não vale. O que está acontecendo? Vamos analisar melhor o que acontece com nossa matriz a cada movimento de um fantasma. No `move_fantasma` vamos imprimir nosso mapa temporário:

```
def move_fantasma(mapa, linha, coluna)
    # ...

    puts "Movendo fantasma encontrado em #{linha} #{coluna}"
    desenha mapa
end
```

Rodamos o jogo e acompanhamos o primeiro movimento:

O laço de mover fantasmas percorre nosso array de `String`s até encontrar nosso primeiro fantasma na décima linha, quarta coluna (`9, 3`). Ele pode andar para baixo, portanto nosso algoritmo de movimento desloca-o para baixo:

```
Movendo fantasma encontrado em 9 3
XXXXXXXXX
X  H    X
X X XXX X
X X X   X
X   X X X
  X     X
 XXX XX X
  X     X
X   X X X
XXX   F X
XXXFXXX X
XXX XXX X
XXX     X
```

Aí o programa continua varrendo a linha, encontra nosso segundo fantasma, na mesma linha, sétima coluna (`6`), que não pode ir para baixo, e acaba por ir para a direita:

```
Movendo fantasma encontrado em 9 6
XXXXXXXXX
X  H    X
X X XXX X
X X X   X
X   X X X
  X     X
 XXX XX X
  X     X
X   X X X
XXX    FX
XXXFXXX X
XXX XXX X
XXX     X
```

Até aqui o algoritmo roda como o esperado. O que acontece a seguir é surpreendente:

```
Movendo fantasma encontrado em 10 3
XXXXXXXXX
X  H    X
X X XXX X
X X X   X
X   X X X
  X     X
 XXX XX X
  X     X
X   X X X
XXX    FX
XXX XXX X
XXXFXXX X
XXX     X
```

O nosso algoritmo procura fantasmas na linha seguinte e encontra o primeiro novamente, movendo-o mais uma vez para baixo! E o processo continua, descendo, descendo:

```
XXXXXXXXX
X  H    X
X X XXX X
X X X   X
X   X X X
  X     X
 XXX XX X
  X     X
X   X X X
XXX    FX
XXX XXX X
XXX XXX X
XXXF    X
```

Acontece que, ao utilizarmos o mesmo pedaço de memória, a mesma matriz, para representar duas coisas, e fazer com que ele possa mudar seu valor, isso é, ele seja mutável (*mutable*), permitimos que nosso algoritmo funcionasse de uma maneira que não desejávamos. Ele altera a matriz de posições atuais, mas ela passa a representar tanto o futuro quanto o passado de nossos fantasmas. Valores mutáveis (e matrizes costumam ser tratadas assim) são perigosos por isso: tome cuidado ao alterá-las.

Como resolver nosso problema? Precisamos separar o mapa, que representa a situação dos fantasmas no início da rodada, da situação do mapa que representa os fantasmas após seus movimentos. Como fazer isso? Podemos criar um novo array de `String`s, só com os muros e os heróis, e copiar os fantasmas um a um, à medida que eles se movem.

Se olharmos a matriz antiga para procurar os fantasmas, vamos encontrá-los somente uma vez, evitando o bug atual.

Mas espera um instante, Guilherme. O fantasma desceu tudo, mas por que ele não foi tudo para a direita? Repare que no fim do turno ele termina em:

```
XXXXXXXXX
X  H    X
X X XXX X
X X X   X
X   X X X
```

```
  X     X
 XXX XX X
  X     X
X   X X X
XXX    FX
XXX XXX X
XXX XXX X
XXX F   X
```

Acontece que, ao invocarmos a função `chars` de nossa `String` para iterarmos com o `each_with_index`, o Ruby tira uma cópia de nosso array de caracteres. Portanto, qualquer mudança que fizermos no array antigo não afeta o novo array. Isso é, de graça já fizemos a cópia que teremos que fazer com as linhas.

Logo, removemos as duas linhas que imprimiam nossas linhas de informação, voltando à versão antiga:

```ruby
def move_fantasma(mapa, linha, coluna)
    posicoes = posicoes_validas_a_partir_de mapa, [linha, coluna]
    if posicoes.empty?
        return
    end

    posicao = posicoes[0]

    mapa[linha][coluna] = " "
    mapa[posicao[0]][posicao[1]] = "F"
end
```

19.3 Copiando nosso mapa

Vamos copiar nosso mapa para resolver o problema dos fantasmas teletransportadores. Ao iniciar o movimento dos fantasmas, copiamos nosso array. Poderíamos fazer dois laços aninhados, dois `for`s passando por cada elemento e copiando-os um a um:

```ruby
def copia_mapa(mapa)
    novo_mapa = []
```

```
    mapa.each do |linha|
        nova_linha = ""
        linha.chars.each do |caractere|
            nova_linha << caractere
        end
        novo_mapa << nova_linha
    end
    novo_mapa
end
```

Precisaríamos ainda verificar se o elemento é um fantasma, removendo-o:

```
def copia_mapa(mapa)
    novo_mapa = []
    mapa.each do |linha|
        nova_linha = ""
        linha.chars.each do |caractere|
            if caractere == "F"
                nova_linha << " "
            else
                nova_linha << caractere
            end
        end
        novo_mapa << nova_linha
    end
    novo_mapa
end
```

Que horror. Em vez disso, podemos utilizar aquela função que já conhecemos, que duplica um valor que a suporte (o `dup`), e outra função que troca, traduz (*tr*), os `F`s por espaços:

```
def copia_mapa(mapa)
    novo_mapa = []
    mapa.each do |linha|
        nova_linha = linha.dup.tr "F", " "
        novo_mapa << nova_linha
    end
    novo_mapa
end
```

Esse código ainda está bem grande. Repare que, se podemos trocar `"F"`s por `" "`, podemos juntar todo o mapa em uma única `String`, fazendo um *join*. O *join* junta diversas `Strings`:

```
nomes = ["guilherme", "de", "azevedo", "silveira"]
puts nomes.join(" ") # guilherme de azevedo silveira
puts nomes.join("\n") # guilherme
# de
# azevedo
# silveira
```

Juntamos nosso mapa em uma única `String`:

```
def copia_mapa(mapa)
  texto = mapa.join("\n")
  # e agora?
end
```

Agora trocamos todos de uma vez, e quebramos novamente:

```
def copia_mapa(mapa)
  novo_mapa = mapa.join("\n").tr("F", " ").split("\n")
end
```

19.4 Movendo os fantasmas na matriz copiada

Nossa função que move os fantasmas precisa agora efetuar as mudanças no novo mapa:

```
def move_fantasmas(mapa)
    caracter_do_fantasma = "F"
    novo_mapa = copia_mapa mapa

    # ...
end
```

Ao mover o fantasma enviaremos tanto o mapa antigo quanto o novo:

```
def move_fantasmas(mapa)
    caracter_do_fantasma = "F"
```

```
    novo_mapa = copia_mapa mapa
    mapa.each_with_index do |linha_atual, linha|
        linha_atual.chars.each_with_index do |caractere_atual,
        coluna|
            eh_fantasma = caractere_atual == caracter_do_fantasma
            if eh_fantasma
                move_fantasma mapa, novo_mapa, linha, coluna
            end
        end
    end
end
```

Nossa função `move_fantasma` deve receber os dois mapas:

```
def move_fantasma(mapa, novo_mapa, linha, coluna)
    posicoes = posicoes_validas_a_partir_de mapa, [linha, coluna]
    if posicoes.empty?
        return
    end

    posicao = posicoes[0]

    mapa[linha][coluna] = " "
    mapa[posicao[0]][posicao[1]] = "F"
end
```

Olhando o código anterior, onde devemos trocar o `mapa` pelo `novo_mapa`? Primeiro avaliamos as posições válidas. Só é válido ir para uma posição se não tem nenhum fantasma lá, e se nenhum se moveu para lá no novo mapa. Portanto, temos que verificar os dois mapas, não só um deles. Passemos os dois mapas como argumento:

```
def move_fantasma(mapa, novo_mapa, linha, coluna)
    posicoes = posicoes_validas_a_partir_de mapa, novo_mapa,
    [linha, coluna]
    if posicoes.empty?
        return
    end
```

```
    posicao = posicoes[0]

    mapa[linha][coluna] = " "
    mapa[posicao[0]][posicao[1]] = "F"
end
```

Alteramos também a função `posicoes_validas_a_partir_de` para verificar se os dois mapas têm uma posição válida:

```
def posicoes_validas_a_partir_de(mapa, novo_mapa, posicao)
    posicoes = []
    baixo = [posicao[0] + 1, posicao[1]]
    if posicao_valida? mapa, baixo && posicao_valida?
    novo_mapa, baixo
        posicoes << baixo
    end
    direita = [posicao[0], posicao[1] + 1]
    if posicao_valida? mapa, direita && posicao_valida?
    novo_mapa, direita
        posicoes << direita
    end
    cima = [posicao[0] - 1, posicao[1]]
    if posicao_valida? mapa, cima && posicao_valida?
    novo_mapa, cima
        posicoes << cima
    end
    esquerda = [posicao[0], posicao[1] - 1]
    if posicao_valida? mapa, esquerda && posicao_valida?
    novo_mapa, esquerda
        posicoes << esquerda
    end
    posicoes
end
```

Um fantasma só pode andar para posições nas quais, no novo mapa, não há nenhum outro fantasma, portanto na primeira invocação a `posicoes_validas_a_partir_de` devemos usar o novo mapa.

Por fim, devemos desenhar o fantasma no novo mapa:

```
def move_fantasma(mapa, novo_mapa, linha, coluna)
    posicoes = posicoes_validas_a_partir_de mapa, novo_mapa,
    [linha, coluna]
    if posicoes.empty?
        return
    end

    posicao = posicoes[0]

    mapa[linha][coluna] = " "
    novo_mapa[posicao[0]][posicao[1]] = "F"
end
```

Mas se alteramos um array dentro da função `move_fantasma`, será que ele também é trocado fora da função? Pensemos no *stack trace*:

```
move_fantasma (mapa = ???, novo_mapa = ???)
move_fantasmas (mapa = ???, novo_mapa = ???)
...
```

Qual é o valor de um array? Já vimos que uma variável que referencia um array é somente um valor, e o array em si está em um único canto da memória. Para um mapa mais simples, teríamos:

```
move_fantasma (mapa = 123, novo_mapa = 456)
move_fantasmas (mapa = 123, novo_mapa = 456)
...
Em um canto da memoria:

123: [XXX]
     [ F ]
     [XXX]
456: [XXX]
     [   ]
     [XXX]
```

Se alterarmos uma posição na referência `456`, ambas possuem algo alterado. Esse algo não é a referência em si, ambas continuam referenciando, apontando, para o mesmo valor da memória:

```
move_fantasma (mapa = 123, novo_mapa = 456)
move_fantasmas (mapa = 123, novo_mapa = 456)
...
Em um canto da memoria:

123: [XXX]
     [ F ]
     [XXX]
456: [XXX]
     [  F]
     [XXX]
```

Devemos sempre nos lembrar: se o valor de uma variável é um valor numérico simples, passá-lo como parâmetro envia o valor em si. Se ele é um valor mais complexo, um objeto, como `String`, `Array` ou outro, temos somente uma referência para esse valor mais complexo na memória.

Testamos o jogo:

```
main.rb:1:in `require_relative': fogefoge.rb:56:
    syntax error, unexpected tIDENTIFIER, expecting keyword_do
                        or '{' or '(' (SyntaxError)
    if posicao_valida? mapa, baixo && posicao_valida? novo_mapa,
    baixo
```

O Ruby se perdeu com nossa invocação à função `posicao_valida`. Note que, como não usamos parênteses para a invocação, ele se perdeu com o `&&` e com a nova invocação de função logo após ele. Como foi mencionado anteriormente, em algumas situações somos obrigados a usar os parênteses. Mas isso já é um ótimo indício de que nosso código está horrível. Atacaremos esse problema a seguir. Agora, corrigimos a invocação:

```
def posicoes_validas_a_partir_de(mapa, novo_mapa, posicao)
    posicoes = []
    baixo = [posicao[0] + 1, posicao[1]]
    if posicao_valida?(mapa, baixo) && posicao_valida?(novo_mapa,
    baixo)
        posicoes << baixo
    end
    direita = [posicao[0], posicao[1] + 1]
```

```
    if posicao_valida?(mapa, direita) &&
    posicao_valida?(novo_mapa, direita)
        posicoes << direita
    end
    cima = [posicao[0] - 1, posicao[1]]
    if posicao_valida?(mapa, cima) && posicao_valida?(novo_mapa,
    cima)
        posicoes << cima
    end
    esquerda = [posicao[0], posicao[1] - 1]
    if posicao_valida?(mapa, esquerda) &&
    posicao_valida?(novo_mapa, esquerda)
        posicoes << esquerda
    end
    posicoes
end
```

Rodamos o código:

```
XXXXXXXXX
X  H    X
X X XXX X
X X X   X
X   X X X
  X     X
 XXX XX X
  X     X
X   X X X
XXX     X
XXX XXX X
XXX XXX X
XXX     X
```

Agora os fantasmas se foram? Claro! Criamos o nosso novo array, de fim de jogada, mas não o trocamos pelo antigo! Quando o turno acaba, devemos dizer que o novo array deve ser utilizado para o próximo turno. Vamos fazer o `move_fantasmas` retornar o novo mapa:

```
def move_fantasmas(mapa)
    caracter_do_fantasma = "F"
```

```
    novo_mapa = copia_mapa mapa
    # ...
    novo_mapa
end
```

Ao jogar, ele é trocado:

```
def joga(nome)
    mapa = le_mapa(2)
    while true
        # ...

        mapa = move_fantasmas mapa
    end
end
```

Rodamos novamente e agora nossos fantasmas nem desaparecem, nem se teletransportam:

```
XXXXXXXXX
X  H    X
X X XXX X
X X X   X
X   X X X
  X     X
 XXX XX X
  X     X
X   X X X
XXX    FX
XXXFXXX X
XXX XXX X
XXX     X
```

19.5 Refatorando o movimento do fantasma

O único problema é que nosso `posicoes_validas_a_partir_de` está horrendo.

```
def posicoes_validas_a_partir_de(mapa, novo_mapa, posicao)
    posicoes = []
```

```
    baixo = [posicao[0] + 1, posicao[1]]
    if posicao_valida?(mapa, baixo) &&
    posicao_valida?(novo_mapa, baixo)
        posicoes << baixo
    end
    direita = [posicao[0], posicao[1] + 1]
    if posicao_valida?(mapa, direita) &&
    posicao_valida?(novo_mapa, direita)
        posicoes << direita
    end
    cima = [posicao[0] - 1, posicao[1]]
    if posicao_valida?(mapa, cima) &&
    posicao_valida?(novo_mapa, cima)
        posicoes << cima
    end
    esquerda = [posicao[0], posicao[1] - 1]
    if posicao_valida?(mapa, esquerda) &&
    posicao_valida?(novo_mapa, esquerda)
        posicoes << esquerda
    end
    posicoes
end
```

Assim como fizemos uma refatoração da movimentação do jogador, esse código parece ter um padrão ao testar cada posição ao redor de nosso fantasma. Podemos criar um array simples com os valores adequados para cada uma das quatro posições:

```
movimentos = [[-1, 0], [0, +1], [+1, 0], [0, -1]]
```

E para cada uma das posições possíveis:

```
movimentos = [[-1, 0], [0, +1], [+1, 0], [0, -1]]
movimentos.each do |movimento|

end
```

Calculamos a nova posição:

```
movimentos = [[-1, 0], [0, +1], [+1, 0], [0, -1]]
movimentos.each do |movimento|
    nova_posicao =
        [posicao[0] + movimento[0], posicao[1] + movimento[1]]
    if posicao_valida?(mapa, nova_posicao) &&
    posicao_valida?(novo_mapa, nova_posicao)

    end
end
```

Verificamos se ela é válida, acumulando-a em um array que será retornado no fim de nossa função:

```
def posicoes_validas_a_partir_de(mapa, novo_mapa, posicao)
    posicoes = []
    movimentos = [[-1, 0], [0, +1], [+1, 0], [0, -1]]
    movimentos.each do |movimento|
        nova_posicao =
           [posicao[0] + movimento[0], posicao[1] + movimento[1]]
        if posicao_valida?(mapa, nova_posicao) &&
        posicao_valida?(novo_mapa, nova_posicao)
            posicoes << nova_posicao
        end
    end
    posicoes
end
```

Podemos extrair também uma função de soma de vetores, que soma as duas posições:

```
def soma(vetor1, vetor2)
    [vetor1[0] + vetor2[0], vetor1[1] + vetor2[1]]
end

def posicoes_validas_a_partir_de(mapa, novo_mapa, posicao)
    posicoes = []
    movimentos = [[-1, 0], [0, +1], [+1, 0], [0, -1]]
    movimentos.each do |movimento|
        nova_posicao = soma posicao, movimento
        if posicao_valida?(mapa, nova_posicao) &&
```

```
        posicao_valida?(novo_mapa, nova_posicao)
            posicoes << nova_posicao
        end
    end
    posicoes
end
```

19.6 O fantasma cavaleiro

E se nosso fantasma se comportar como uma peça do tipo cavalo no jogo de xadrez? Isso é, em vez de ir para cima, baixo, esquerda e direita, queremos que ele faça movimentos do tipo 'L'. O exemplo a seguir mostra a posição do fantasma e marcando com * as oito posições para onde ele poderia ir após um único movimento:

```
XXXXXXX
X * * X
X     X
X*   *X
X  F  X
X*   *X
X     X
X * * X
XXXXXXX
```

Como implementar essa lógica? Note que são oito movimentos válidos:

```
movimentos = [[-2, -1], [-2, +1], [+2, -1], [+2, +1],
              [-1, -2], [-1, +2], [+1, -2], [+1, +2]]
```

Pronto, os fantasmas agora andam como cavalos do xadrez. Com esse tipo de técnica simples conseguimos inclusive implementar movimentos complexos de peças de tabuleiro em espaços 3d! Bastaria adicionar a terceira dimensão. Claro, no nosso jogo o fantasma não é um cavalo, mas se quiser dificultar o jogo depois, pode alterar os movimentos válidos do fantasma facilmente.

19.7 Movimento aleatório dos fantasmas

Precisamos escolher uma das posições, aleatoriamente. Já sabemos fazê-lo com o uso do `rand`:

```
def move_fantasma(mapa, novo_mapa, linha, coluna)
    posicoes = posicoes_validas_a_partir_de mapa, novo_mapa,
    [linha, coluna]
    if posicoes.empty?
        return
    end

    aleatoria = rand posicoes.size
    posicao = posicoes[aleatoria]

    mapa[linha][coluna] = " "
    novo_mapa[posicao[0]][posicao[1]] = "F"
end
```

Pronto. Podemos testar o jogo duas vezes e ver que cada vez os fantasmas se locomovem de maneira diferente. Note que sua saída pode ser (e provavelmente será) diferente da minha. Começando o jogo e movendo o herói uma vez:

```
XXXXXXXXX
X  H    X
X X XXX X
X X X   X
X   X X X
  X     X
 XXX XX X
  X     X
X  FX X X
XXX  F  X
XXX XXX X
XXX XXX X
XXX     X
```

Começando novamente o jogo, movendo uma primeira vez:

```
XXXXXXXXX
X  H    X
X X XXX X
X X X   X
X   X X X
  X     X
 XXX XX X
  X     X
X  FX X X
XXX    FX
XXX XXX X
XXX XXX X
XXX     X
```

19.8 Quando o herói perde

Outra situação importante que ainda não definimos é quando o nosso herói entra em contato com um fantasma. Isso pode ocorrer quando o turno acaba e tanto o fantasma quanto o herói estão na mesma posição. Mas como descobrir que isso ocorreu?

Como o fantasma é sempre o último a andar, ele vai aparecer "por cima" do herói: o jogador sumirá do mapa. Portanto, se não encontramos o jogador no mapa, sabemos que ele perdeu o jogo.

Nossa função que encontra um jogador deve agora retornar que não encontrou, vazio, nulo:

```
def encontra_jogador(mapa)
    caracter_do_heroi = "H"
    mapa.each_with_index do |linha_atual, linha|
        coluna_do_heroi = linha_atual.index caracter_do_heroi
        if coluna_do_heroi
            return [linha, coluna_do_heroi]
        end
    end
    nil
end
```

Ao terminar o turno, verificamos se o jogador não está mais no mapa e,

se for o caso, mostramos a mensagem de *Game Over*.

```
def joga(nome)
    mapa = le_mapa(2)
    while true
        # ...
        mapa = move_fantasmas mapa
        if encontra_jogador(mapa) == nil
            game_over
            break
        end
    end
end
```

Em Ruby, verificar se um valor é nulo é a mesma coisa que verificar que ele não existe:

```
def joga(nome)
    mapa = le_mapa(2)
    while true
        # ...
        mapa = move_fantasmas mapa
        if !encontra_jogador(mapa)
            game_over
            break
        end
    end
end
```

Mas nossa função está bem feia. Novamente temos uma linha que faz muita coisa: `if`, `!` e invoca uma função. Vamos extrair o código, lembre-se que a variável é desnecessária e só é utilizada para deixar claro o que estamos retornando:

```
def jogador_perdeu?(mapa)
    perdeu = !encontra_jogador(mapa)
end

def joga(nome)
    mapa = le_mapa(2)
```

```
    while true
        # ...
        mapa = move_fantasmas mapa
        if jogador_perdeu?(mapa)
            game_over
            break
        end
    end
end
```

Criamos também a função de `game_over` na nossa interface:

```
def game_over
    puts "\n\n\n\n\n\n"
    puts "Game Over"
end
```

Pronto, podemos jogar até o momento que perdemos:

```
XXXXXXXXX
X       X
X X XXX X
X X X   X
X   X X X
  X     X
 XXX XX X
 FX F   X
X  HX X X
XXX     X
XXX XXX X
XXX XXX X
XXX     X
Para onde deseja ir?
W






Game Over
```

Como o movimento de dois fantasmas é aleatório, demora para perdemos.

19.9 Retorno nulo ou opcional?

Nossa função `encontra_jogador` retorna nada caso ele não encontre um jogador no mapa. O `nil` é muito útil para marcar que uma função que retorna algo pode, sob determinadas circunstâncias, retornar nada. O problema acontece ao usar esse retorno. Como o desenvolvedor pode escrever código que invoca a função e faz qualquer coisa com o retorno, ele não é obrigado a verificar se o valor retornado é vazio. Se não for vazio, a aplicação pode parar com um erro ou ainda se comportar de maneira totalmente inesperada.

Algumas linguagens de programação funcionais como Swift apresentam um conceito de opcional, outras linguagens orientadas a objeto como Java oferecem tal funcionalidade, que já obriga o desenvolvedor a verificar se o retorno é válido antes de utilizá-lo.

19.10 Resumindo

Vimos como é importante cuidar direitinho de nossas referências na memória. Um valor como um array pode ser alterado por fora e por dentro de uma função, causando bagunça naquilo que queremos representar. Em jogos baseados em turno (*turn-based*), assim como em processos que devem executar algo baseado em um estado, é comum que o estado seja representado por um valor e a cada novo estado tiremos uma cópia inteira dele. Podemos até voltar ao estado anterior (*undo*) com tal prática que tem nome, *Memento*! O cuidado que devemos tomar é com a memória, claro.

Implementamos o movimento aleatório e o fim do jogo quando o herói perde para um fantasma. Em diversos desses instantes utilizamos práticas típicas de um programador de jogos ou de maratona de programação, onde usamos arrays para trabalhar como atalhos para determinados resultados.

Como veremos adiante, estruturas, classes e objetos podem representar o mesmo tipo de informação, mas com mais valor de significado (um mapa não é um array, é um mapa um herói não é um array de tamanho 2, é um herói

em uma linha e uma coluna), algo que pode ser benéfico para nosso código em longo prazo.

Capítulo 20

Estruturas e classes: uma introdução a Orientação a Objetos

20.1 A bagunça dos defs

Olhemos nossa função soma, que soma dois vetores para calcular uma nova posição:

```
def soma(vetor1, vetor2)
    [vetor1[0] + vetor2[0], vetor1[1] + vetor2[1]]
end
```

Por mais que ela seja uma função que faz sentido, o que ela tem em comum com a função `copia_mapa`?

```
def copia_mapa(mapa)
    mapa.join("\n").tr("F", " ").split("\n")
end
```

São lógicas de negócio, nada mais. Seguindo somente esse critério de separação de lógica de negócios, é fácil imaginar que um projeto ou jogo grande vai ficar com "infinitas" funções juntas. Mais ainda, quem sabe como somar um movimento a uma posição é o gerenciamento de posições. Quem sabe copiar um mapa é o próprio mapa. Nesse instante, todas essas responsabilidades estão no mesmo lugar, jogadas em um único arquivo. Até mesmo o herói! Como assim o herói é um array de duas posições? O herói é um herói, é você, sou eu. Ele não é um mero array.

Apesar de nosso foco ser uma introdução a computação e programação, por utilizarmos a linguagem Ruby como exemplo, desenvolveremos um pouco mais nosso jogo no sentido de fazê-lo suportar o mínimo de orientação a objetos... e começaremos pelo que há de mais importante para nós, o jogador, nosso herói.

20.2 Extraindo uma primeira estrutura

Até agora representamos um herói com um array de tamanho dois, o que é algo muito estranho. Pensando em valores, um herói é um herói, não um array. Para movê-la, basta mudar sua linha e coluna, mandar se movimentar. Seria interessante criar um único valor, `Heroi`, que tivesse dois valores dentro dele, a `linha` e a `coluna` atual.

Queremos criar uma estrutura que representa um herói, uma abstração de nosso herói. Essa abstração não é nosso herói de verdade, somente uma definição de como um herói se comporta no mundo real. Vamos criar uma classe:

```
class Heroi
end
```

Agora podemos criar um herói:

```
class Heroi
end
```

```
heroi = Heroi.new
```

Mas um herói sem linha nem coluna não tem graça. Queremos dizer que um herói tem dois atributos, sua linha e sua coluna:

```
class Heroi
    attr_accessor :linha, :coluna
end
heroi = Heroi.new

heroi.linha = 3
heroi.coluna = 6

puts heroi.linha # 3
puts heroi.coluna # 6
```

Teremos uma variável chamada `heroi` que tem um valor. Esse valor é a referência para um objeto, uma instância de `Heroi`, sendo que dentro desse objeto teremos os valores `linha` e `coluna`. No caso do nosso herói, esses valores são `3` e `6`.

Da mesma maneira que criamos um herói, um objeto do tipo herói, podemos criar dois. Cada um tem valores diferentes, pois tem espaços diferentes na memória:

```
class Heroi
    attr_accessor :linha, :coluna
end

guilherme = Heroi.new
guilherme.linha = 3
guilherme.coluna = 6

paulo = Heroi.new
paulo.linha = 5
paulo.coluna = 3

puts guilherme.linha # 3
puts guilherme.coluna # 6
```

```
puts paulo.linha # 5
puts paulo.coluna # 3
```

O que fizemos até agora? Definimos uma abstração, como funciona um herói, uma classe `Heroi`. Ela dá as regras de como um herói pode ser criado (instanciado), quais seus atributos (variáveis de instância), comportamentos (funções) entre outros. Para nós já é ótimo saber que nosso herói no sistema agora é realmente um herói, e não uma gambiarra de um vetor. Portanto, o código final do arquivo `heroi.rb` é:

```
class Heroi
    attr_accessor :linha, :coluna
end
```

20.3 Usando uma estrutura

Devemos usar nosso jogador como uma instância de `Heroi`, para isso alteraremos o momento em que ele é encontrado, o `encontra_jogador`:

```
def encontra_jogador(mapa)
    caracter_do_heroi = "H"
    mapa.each_with_index do |linha_atual, linha|
        coluna_do_heroi = linha_atual.index caracter_do_heroi
        if coluna_do_heroi
            return [linha, coluna_do_heroi]
        end
    end
    nil
end
```

Quando encontramos um jogador, instanciamos, criamos um novo objeto do tipo `Heroi` na linha e na coluna adequada:

```
def encontra_jogador(mapa)
    caracter_do_heroi = "H"
    mapa.each_with_index do |linha_atual, linha|
        coluna_do_heroi = linha_atual.index caracter_do_heroi
        if coluna_do_heroi
```

```
            heroi = Heroi.new
            heroi.linha = linha
            heroi.coluna = coluna_do_heroi
            return heroi
        end
    end
    nil
end
```

Já que usamos a classe `Heroi`, precisamos carregá-la:

```
require_relative 'ui'
require_relative 'heroi'

# ...
```

Tentamos encontrar o jogador na função `joga`:

```
heroi = encontra_jogador mapa
nova_posicao = calcula_nova_posicao heroi, direcao
if !posicao_valida? mapa, nova_posicao
    next
end

mapa[heroi[0]][heroi[1]] = " "
mapa[nova_posicao[0]][nova_posicao[1]] = "H"
```

Primeiro a usamos para encontrar a posição do herói no mapa, o que podemos mudar para:

```
mapa[heroi.linha][heroi.coluna] = " "
```

O herói também é passado como argumento para o `calcula_nova_posicao`:

```
nova_posicao = calcula_nova_posicao heroi, direcao
```

Que brinca com os valores `[0]` e `[1]` de nosso antigo array.

```
def calcula_nova_posicao(heroi, direcao)
    heroi = heroi.dup
    movimentos = {
        "W" => [-1, 0],
        "S" => [+1, 0],
        "A" => [0, -1],
        "D" => [0, +1]
        }
    movimento = movimentos[direcao]
    heroi[0] += movimento[0]
    heroi[1] += movimento[1]
    heroi
end
```

Isso não faz mais sentido, afinal um herói não tem mais esses acessos. Ele possui uma linha e uma coluna, logo:

```
def calcula_nova_posicao(heroi, direcao)
    heroi = heroi.dup
    movimentos = {
        "W" => [-1, 0],
        "S" => [+1, 0],
        "A" => [0, -1],
        "D" => [0, +1]
        }
    movimento = movimentos[direcao]
    heroi.linha += movimento[0]
    heroi.coluna += movimento[1]
    heroi
end
```

20.4 Code smell: feature envy

Opa. Nosso `calcula_nova_posicao` tem algo de estranho. Das 6 instruções contidas no código, 4 delas (dois terços!) envolvem nosso herói! O código nos indica que a função `calcula_nova_posicao` está com inveja do herói, ela queria ter as funcionalidades dele e por isso fica perguntando cinquenta coisas para ele: `dup` por favor, `linha` por favor, `coluna` por

favor, `retorna você` por favor. Quanta inveja, inveja de funcionalidade, *feature envy*.

Essa característica é um fedor de código que indica que essa função poderia pertencer a quem ela tanto "gosta". Afinal, perguntemos a nós mesmos, quem entende mais de um personagem heroico do que ele mesmo? Quem sabe movimentar um herói? Ele mesmo! Além disso, não faz nenhum sentido a função `calcula_nova_posicao` ser invocada se um herói não existe. Por todos esses motivos, podemos mover a função para dentro de nosso `Heroi`:

```
class Heroi
    attr_accessor :linha, :coluna
    def calcula_nova_posicao(heroi, direcao)
        heroi = heroi.dup
        movimentos = {
            "W" => [-1, 0],
            "S" => [+1, 0],
            "A" => [0, -1],
            "D" => [0, +1]
            }
        movimento = movimentos[direcao]
        heroi.linha += movimento[0]
        heroi.coluna += movimento[1]
        heroi
    end
end
```

Mas se a função já está dentro do próprio herói, não precisamos recebê-lo como argumento. Vamos remover o argumento:

```
class Heroi
    attr_accessor :linha, :coluna
    def calcula_nova_posicao(direcao)
        # ...
    end
end
```

Já que estamos rodando esse código dentro de nosso herói, e não o temos mais como argumento, a variável `heroi` não existe mais para executarmos

o `dup`. Como invocar o método `dup` no próprio objeto? Como dizer para o código de um método dentro de um objeto para invocar outro método nele mesmo, nele próprio? Usamos a palavra "próprio", em inglês *self*:

```
class Heroi
    attr_accessor :linha, :coluna
    def calcula_nova_posicao(direcao)
        heroi = self.dup
        movimentos = {
            "W" => [-1, 0],
            "S" => [+1, 0],
            "A" => [0, -1],
            "D" => [0, +1]
            }
        movimento = movimentos[direcao]
        heroi.linha += movimento[0]
        heroi.coluna += movimento[1]
        heroi
    end
end
```

Por fim, quando invocamos um método em nós mesmos, o uso do `self` é na verdade opcional. Podemos:

```
class Heroi
    attr_accessor :linha, :coluna
    def calcula_nova_posicao(direcao)
        heroi = dup
        movimentos = {
            "W" => [-1, 0],
            "S" => [+1, 0],
            "A" => [0, -1],
            "D" => [0, +1]
            }
        movimento = movimentos[direcao]
        heroi.linha += movimento[0]
        heroi.coluna += movimento[1]
        heroi
    end
end
```

Nossa função `joga` deve agora invocar o método no próprio objeto:

```
def joga(nome)
    mapa = le_mapa(2)
    while true
        desenha mapa
        direcao = pede_movimento

        heroi = encontra_jogador mapa
        nova_posicao = heroi.calcula_nova_posicao direcao
        if !posicao_valida? mapa, nova_posicao
            next
        end

        mapa[heroi.linha][heroi.coluna] = " "
        mapa[nova_posicao[0]][nova_posicao[1]] = "H"

        mapa = move_fantasmas mapa
        if jogador_perdeu?(mapa)
            game_over
            break
        end
    end
end
```

20.5 Boa prática: buscar quem invoca antes de refatorar

Agora nossa `nova_posicao` também é um objeto do tipo `Heroi`. Ela é usada em nossa função também ao substituirmos o valor dela no mapa, o que já sabemos alterar para utilizar os atributos `linha` e `posicao`:

```
mapa[nova_posicao.linha][nova_posicao.coluna] = "H"
```

Mas ela também é utilizada para verificar se a nova posição é válida:

```
if !posicao_valida? mapa, nova_posicao
    next
end
```

Antes de refatorarmos o código de uma função *sempre* devemos olhar quem a invoca. Nesse caso, tanto aqui quanto no momento de movimentar um fantasma invocamos a função `posicao_valida?`. Isso significa que, se alterarmos o código dela para suportar um `Heroi` em vez de um array de duas posições, quebraremos o código do fantasma, e não queremos isso nesse instante de nossa refatoração. O que queremos é fazer essa invocação funcionar.

Como transformar um `Heroi` em um array de duas posições para poder continuar invocando um método que funciona tanto para o herói quanto para o fantasma?

Podemos criar um método novo, seguindo o padrão `to_i` do Ruby, algo como `to_array` (em vez de `to_a` que é ambíguo demais):

```
class Heroi
    attr_accessor :linha, :coluna
    def calcula_nova_posicao(heroi, direcao)
        # ...
    end

    def to_array
        [linha, coluna]
    end
end
```

Terminamos com nosso herói:

```
class Heroi
    attr_accessor :linha, :coluna
    def calcula_nova_posicao(direcao)
        heroi = dup
        movimentos = {
            "W" => [-1, 0],
            "S" => [+1, 0],
            "A" => [0, -1],
            "D" => [0, +1]
            }
        movimento = movimentos[direcao]
        heroi.linha += movimento[0]
```

```
        heroi.coluna += movimento[1]
        heroi
    end

    def to_array
        [linha, coluna]
    end
end
```

E invocamos o método `to_array` ao verificar se a nova posição é válida, ficando com o método `joga`:

```
def joga(nome)
    mapa = le_mapa(2)
    while true
        desenha mapa
        direcao = pede_movimento

        heroi = encontra_jogador mapa
        nova_posicao = heroi.calcula_nova_posicao direcao
        if !posicao_valida? mapa, nova_posicao.to_array
            next
        end

        mapa[heroi.linha][heroi.coluna] = " "
        mapa[nova_posicao.linha][nova_posicao.coluna] = "H"

        mapa = move_fantasmas mapa
        if jogador_perdeu?(mapa)
            game_over
            break
        end
    end
end
```

Pronto. Nosso jogo já está funcionando. Agora com nosso herói bem representado, e parte do código que lhe pertence (`calcula_nova_posicao`) dentro dele.

20.6 Boa prática: Tell, don't ask

Novamente temos repetição de código. Dado um herói, fazemos:

```
mapa[heroi.linha][heroi.coluna] = " "
```

Depois, dada uma nova posição, marcamos com `H`:

```
mapa[nova_posicao.linha][nova_posicao.coluna] = "H"
```

Em vez de pedir informação (*ask*), parece fazer mais sentido mandar executar algo (*tell*):

```
heroi.remove_do mapa
nova_posicao.coloca_no mapa
```

O padrão que estamos adotando é o de mandar executar algo, ao invés de perguntar: *tell, don't ask*. Não nos importa se o herói é implementado com `linha` e `coluna`, `x` e `y`, array de tamanho dois. Não importa. Importa que uma posição é capaz de se remover de um mapa. E que ela é capaz de se colocar no mapa.

Vamos definir o método `remove_do`, o ato de fazer um *extract method*:

```
def remove_do(mapa)
    mapa[linha][coluna] = " "
end
```

E o `coloca_no`:

```
def coloca_no(mapa)
    mapa[linha][coluna] = "H"
end
```

20.7 Atributos e attr_accessor

O *attr_accessor* na verdade cria dois métodos. Um para alterar e outro para ler o valor de um atributo. Como ler um atributo diretamente de dentro de uma classe? Através de um `@`, como no caso:

```
def coloca_no(mapa)
    mapa[@linha][@coluna] = "H"
end
```

Como estamos utilizando um `attr_accessor`, tanto faz mantermos o `@` ou não; manteremos sem o `@`.

Além do `attr_accessor`, existem também o `attr_reader`, que fornece somente um leitor, e o `attr_writer`, que disponibiliza somente o método de escrita.

20.8 Estrutura ou Classe?

Costumamos chamar de *estrutura* a definição de um conjunto de valores agrupados. No começo do nosso código, utilizamos um `Heroi` para somente agrupar valores, nada mais. Costumamos chamar os valores de uma estrutura de "dados" ou "estrutura", mas cuidado para não confundir a instância dessa estrutura (com dados) com a definição da estrutura.

Uma classe é a definição de um agrupamento tanto de valores como de comportamentos, funções. Nesse caso chamamos essas funções de *métodos*, como vimos antes. Ao criarmos uma instância de uma classe, chamamos esse valor de objeto.

Já havíamos utilizados objetos antes. Na realidade, todo valor em Ruby é um objeto. As *String*s, *Array*s, *Fixnum* e *Float* são todos tipos (classes) dos quais foram criadas instâncias. Por exemplo, ao usarmos o valor `15` estamos com uma instância de `Fixnum` cujo valor interno é `15`. Ao criarmos um array de números temos na verdade um objeto (o array) que aponta para diversos objetos (cada um dos números).

Dependendo da linguagem, a definição varia um pouco, não se assuste ao aprender uma nova linguagem. Uma funcionalidade de uma pode ter um nome um pouco diferente em outra.

Code smell: classes anêmicas

Aqui em Ruby, acabamos por usar basicamente classes como classes de verdade, e não como meras estruturas bobas (comumente chamadas de classes anêmicas).

20.9 A verdade por trás de métodos, funções e lambdas

Já que uma função é um nome para um trecho de código, podemos pensar que uma função também poderia ser um tipo de variável, assim como um array é. Na prática é isso que acontece! Por trás dos panos, tanto uma função quanto um valor qualquer é referenciado através de um nome, o nome da função ou da variável.

Só que em Ruby não conseguimos fazer diretamente uma variável referenciar uma função que acaba de ser definida como vimos até agora, devido à regra dos parênteses. Em Javascript isso seria possível:

```
function bemvindo(nome) {
    println("Bem vindo " + nome + "!");
}

var minhaFuncao = bemvindo;
minhaFuncao("Guilherme");
```

Em Ruby, o código análogo invocaria a função, sem passar parâmetros, ao tentar referênciá-la:

```
def bemvindo(nome)
  puts "Bem vindo " + nome + "!"
end

minhaFuncao = bemvindo # tenta invocar a função sem parâmetro!
minhaFuncao "Guilherme" # não faz sentido nenhum
```

Para fazer isso, Ruby permite a criação de funções "soltas", literalmente uma função, chamadas nesse contexto de *lambda literal*, uma função que não está atrelada a nenhum objeto:

```
bemvindo = -> (nome) {
  puts "Bem-vindo " + nome + "!"
}
```

Podemos agora invocá-la, com uma sintaxe bem feia vendo por outro lado, mais educada deixando claro que estamos invocando (*call*) algo:

```
bemvindo.("Guilherme") # Bem vindo Guilherme!
bemvindo.call("Guilherme") # Bem vindo Guilherme!
```

Ou ainda reatribuí-la a outra variável:

```
minhaFuncao = bemvindo
minhaFuncao.("Guilherme") # Bem vindo Guilherme!
```

Como o método é a invocação de uma função no objeto, temos em Ruby que sempre estamos invocando métodos. O conceito de função pura não faz sentido, uma vez que até mesmo o *lambda literal* é um objeto onde chamamos um método, o método `call`. Portanto, não se preocupe: isso não significa que Ruby é ou não é funcional, somente que todo valor em Ruby é um objeto e que as funções que invocamos são chamadas de métodos.

Como vício de linguagem, acabamos usando a palavra *função* para dizer que estamos invocando ou definindo esses métodos, afinal uma função está mais próxima da definição formal disso que estamos invocando. Novamente, não se preocupe, ao conversar com seus colegas use o vocabulário que deixe claro o que é que está fazendo.

Em Ruby, definimos e invocamos métodos mas se você chamá-los de função, ninguém sai machucado.

20.10 Resumindo

Vimos como criar uma estrutura que basicamente armazena dados, mas em Ruby ela já é uma classe, com métodos e atributos. Aprendemos a utilizá-la, instanciando um objeto a partir da definição descrita de nossa classe.

Ao notar que nosso código utilizava muitos valores da estrutura, movemos o comportamento para dentro dela, criando nosso primeiro método. Aprendemos o que significa invocar um método para mandar fazer algo, em vez de perguntar por alguma informação, favorecendo o encapsulamento da lógica por trás do comportamento e incluindo seus atributos. Vimos como expor, acessando e escrevendo valores em atributos.

Outras linguagens podem definir os dois tipos bem distintos: estruturas que costumam somente agrupar valores, e classes que agrupam valores e comportamentos.

Por fim, vimos que todo valor em Ruby é de alguma forma um objeto, portanto sempre que utilizamos o conceito de funções estávamos falando, na verdade, de métodos conceitualmente não há nada de errado em chamá-las de funções, mas em Ruby tais funções são também métodos. Por fim, vimos que se um número e uma `String` podem ser valores a que damos nomes (aplicamos a variáveis), funções também podem ser tratadas assim através de lambdas.

Capítulo 21

Destruindo os fantasmas: o mundo da recursão

21.1 Destruindo os fantasmas

Nosso próximo passo é colocar uma bomba no meio jogo: quando o usuário passa por ela, ele detona os fantasmas próximos. Primeiro criamos um novo mapa, contendo as bombas (*) e mais fantasmas, nosso `mapa3.txt`:

```
XXXXXXXXXXXXXXXXX
X   FFF  X      X
X X XX X X X XX X
X X X* X     X  X
X   X XXXX XXX XX
  X     XX  X   X
 XXX XX XXX X X X
```

```
  X         X X X
X   X X XXX FFF X
X X     XXX XXX X
X X XXX   X XXX X
X  HXXX X X   X X
XXX*FXXFX   X   X
```

Passamos a carregar o terceiro mapa:

```
mapa = le_mapa(3)
```

Primeiro, caso um fantasma entre em contato com uma bomba, ele simplesmente a come, e nosso herói tem menos chances de ganhar o jogo. Portanto, preste atenção, o jogador deve correr atrás das bombas o mais rápido possível.

Já quando o nosso herói encosta em uma bomba, devemos explodir quatro posições para a direita de onde a bomba está. Qualquer fantasma na região deve desaparecer, assim como muros:

```
def joga(nome)
# ...

    heroi.remove_do mapa
    if mapa[nova_posicao.linha][nova_posicao.coluna] == "*"
        mapa[nova_posicao.linha][nova_posicao.coluna + 1] = " "
        mapa[nova_posicao.linha][nova_posicao.coluna + 2] = " "
        mapa[nova_posicao.linha][nova_posicao.coluna + 3] = " "
        mapa[nova_posicao.linha][nova_posicao.coluna + 4] = " "
    end
    nova_posicao.coloca_no map
# ...
end
```

Uma implementação inicial, mas muito ruim. Obviamente podemos trocar por um `for`:

```
heroi.remove_do mapa
if mapa[nova_posicao.linha][nova_posicao.coluna] == "*"
    for direita in 1..4
```

```
        mapa[nova_posicao.linha][nova_posicao.coluna + direita]
        " "
    end
end
nova_posicao.coloca_no mapa
```

Testamos nosso jogo com o terceiro mapa, andando para baixo, e vemos o resultado, quando os dois fantasmas e os muros são destruídos:

```
...
X X XXX   X XXX X
X  HXXX X X   X X
XXX*FXXFX   X   X
Para onde deseja ir?
...
X X XXX   X XXX X
X   XXX X X   X X
XXXH    X   X   X
```

21.2 Andando para a direita

Podemos extrair uma função para este trecho do código, algo que remova tudo do mapa a partir desta posição, por quatro casas. Vamos invocá-la:

```
heroi.remove_do mapa
if mapa[nova_posicao.linha][nova_posicao.coluna] == "*"
    remove mapa, nova_posicao, 4
end
nova_posicao.coloca_no mapa
```

E extrair seu código:

```
def remove mapa, posicao, quantidade
    for direita in 1..quantidade
        mapa[posicao.linha][posicao.coluna + direita] = " "
    end
end
```

O resultado ainda é o mesmo. Mas repare que está estranha essa história de toda vez somar um número cada vez maior. Em vez disso, poderíamos

falar para a posição andar para a direita. Lembra que nosso herói sabe se movimentar? Ele mesmo tem o método `calcula_nova_posicao`:

```
def remove mapa, posicao, quantidade
    for direita in 1..quantidade
        posicao = posicao.calcula_nova_posicao "D"
        mapa[posicao.linha][posicao.coluna] = " "
    end
end
```

Mas, Guilherme, ainda está um pouco estranho. Passar `D` como argumento? Estamos programando orientado a `String` ou a objetos? Seria mais educado fazer:

```
def remove mapa, posicao, quantidade
    for direita in 1..quantidade
        posicao = posicao.direita
        mapa[posicao.linha][posicao.coluna] = " "
    end
end
```

E criarmos o método `direita`:

```
class Heroi
    # ...

    def direita
        calcula_nova_posicao "D"
    end
end
```

21.3 Recursão infinita

Mas repare que a lógica de nossa função ainda é razoavelmente complexa. Note que o que o laço faz é passar com a `direita` do `1` até `4`, isso é, `posicao.coluna+1` até `posicao.coluna+4`. Estamos fazendo um laço, mudando a posição e limpando um trecho do mapa. Não seria possível simplificar nossa função, removendo esse laço?

```
def remove mapa, posicao, quantidade
    for direita in 1..quantidade
        posicao = posicao.direita
        mapa[posicao.linha][posicao.coluna] = " "
    end
end
```

O que seria de nossa função se ela somente removesse a primeira posição da direita?

```
def remove mapa, posicao, quantidade
    posicao = posicao.direita
    mapa[posicao.linha][posicao.coluna] = " "
end
```

Não seria o suficiente, pois ainda temos que remover mais posições à direita, claro. Quantas posições faltam remover? Três. Isso é, `quantidade - 1`. Invoquemos a própria função novamente, agora passando `quantidade - 1`, ou seja, `3`:

```
def remove mapa, posicao, quantidade
    posicao = posicao.direita
    mapa[posicao.linha][posicao.coluna] = " "
    remove mapa, posicao, quantidade - 1
end
```

O que acontece agora? A função será invocada para a segunda posição à direita, com quantidade valendo 2; a próxima posição à direita com quantidade valendo 1; e a última posição à direita com quantidade valendo 0, mas ela continua sendo invocada eternamente!

Vejamos como funciona, vamos simular nossa pilha de execução. Minhas linhas da função `remove` são como a seguir:

```
def remove mapa, posicao, quantidade
    posicao = posicao.direita                          # linha 109
    mapa[posicao.linha][posicao.coluna] = " "    # linha 110
    remove mapa, posicao, quantidade - 1        # linha 111
end
```

Primeiro, chamamos a função `remove` com o `mapa`, com a posição por exemplo `5, 7`, e `quantidade` valendo `4`:

```
fogefoge.rb:remove:109 (mapa =..., posicao = 5,7, quantidade = 4)
```

Ao terminar a execução da linha 109, indo para a 110, temos que a posição foi alterada:

```
fogefoge.rb:remove:110 (mapa =..., posicao = 5,8, quantidade = 4)
```

Portanto, limpamos a posição `5,8` do mapa, o primeiro quadrado à direita. Agora executamos a linha 111, que invoca a função `remove` com `quantidade` valendo 3:

```
fogefoge.rb:remove:109 (mapa =..., posicao = 5,8, quantidade = 3)
fogefoge.rb:remove:111 (mapa =..., posicao = 5,8, quantidade = 4)
```

Primeiro, ele move a posição para a direita:

```
fogefoge.rb:remove:110 (mapa =..., posicao = 5,9, quantidade = 3)
fogefoge.rb:remove:111 (mapa =..., posicao = 5,8, quantidade = 4)
```

Limpa a posição `5,9` e depois chama a função `remove` novamente, empilhando mais uma vez a função:

```
fogefoge.rb:remove:109 (mapa =..., posicao = 5,9, quantidade = 2)
fogefoge.rb:remove:111 (mapa =..., posicao = 5,9, quantidade = 3)
fogefoge.rb:remove:111 (mapa =..., posicao = 5,8, quantidade = 4)
```

Mais uma vez será movido para a direita:

```
fogefoge.rb:remove:110 (mapa =..., posicao = 5,10,
quantidade = 2)
fogefoge.rb:remove:111 (mapa =..., posicao = 5,9, quantidade = 3)
fogefoge.rb:remove:111 (mapa =..., posicao = 5,8, quantidade = 4)
```

O mapa na posição `5,10` é limpo, e invocamos a função novamente:

```
fogefoge.rb:remove:109 (mapa =..., posicao = 5,10,
quantidade = 1)
fogefoge.rb:remove:111 (mapa =..., posicao = 5,10,
quantidade = 2)
fogefoge.rb:remove:111 (mapa =..., posicao = 5,9, quantidade = 3)
fogefoge.rb:remove:111 (mapa =..., posicao = 5,8, quantidade = 4)
```

O mesmo processo ocorre para `5,11`:

```
fogefoge.rb:remove:109 (mapa =..., posicao = 5,11,
quantidade = 0)
fogefoge.rb:remove:111 (mapa =..., posicao = 5,11,
quantidade = 1)
fogefoge.rb:remove:111 (mapa =..., posicao = 5,10,
quantidade = 2)
fogefoge.rb:remove:111 (mapa =..., posicao = 5,9, quantidade = 3)
fogefoge.rb:remove:111 (mapa =..., posicao = 5,8, quantidade = 4)
```

E o fluxo continua...

```
fogefoge.rb:remove:109 (mapa =..., posicao = ...,
quantidade = -576)
...
fogefoge.rb:remove:111 (mapa =..., posicao = 5,13,
quantidade = -1)
fogefoge.rb:remove:111 (mapa =..., posicao = 5,12,
quantidade = 0)
fogefoge.rb:remove:111 (mapa =..., posicao = 5,11,
quantidade = 1)
fogefoge.rb:remove:111 (mapa =..., posicao = 5,10,
quantidade = 2)
fogefoge.rb:remove:111 (mapa =..., posicao = 5,9, quantidade = 3)
fogefoge.rb:remove:111 (mapa =..., posicao = 5,8, quantidade = 4)
```

Podemos ver isso acontecer rodando o jogo:

```
heroi.rb:4: stack level too deep (SystemStackError)
```

Opa. O que aconteceu? O nível de profundidade foi muito grande, pois chamamos uma função, que chamou a si mesma, que chamou a si mesma, que chamou a si mesma, eternamente. Mas claro, existe algum limite como tudo em um computador para o tamanho da pilha, e esse limite alguma hora foi alcançado. O programa parou.

O que estamos fazendo é que nossa função chama a si mesma, essa técnica é chamada de *recursão*.

21.4 A base da recursão

O problema de uma recursão infinita é que uma hora a pilha de execução estoura, e o programa para. Queremos definir um ponto de parada; no nosso caso, só queremos executar nossa função enquanto a quantidade for maior que zero:

```
def remove mapa, posicao, quantidade
    if quantidade == 0
        return
    end
    posicao = posicao.direita
    mapa[posicao.linha][posicao.coluna] = " "
    remove mapa, posicao, quantidade - 1
end
```

Unless: trava de segurança (safeguard)

Outra maneira de escrever esse código seria usando o `unless` de uma linha

```
def remove mapa, posicao, quantidade
    return unless quantidade > 0
    posicao = posicao.direita
    mapa[posicao.linha][posicao.coluna] = " "
    remove mapa, posicao, quantidade - 1
end
```

Temos uma inversão, uma negação, o que pode ficar confuso: não execute isso se for maior do que zero.

Esse ponto de parada da recursão é o que chamamos de base da recursão. Quando ela chega àquele ponto, ela para. Rodando agora nossa aplicação teríamos o seguinte *stack trace*:

```
fogefoge.rb:remove:109 (mapa =..., posicao = 5,12,
quantidade = 0)
fogefoge.rb:remove:112 (mapa =..., posicao = 5,11,
```

```
quantidade = 1)
fogefoge.rb:remove:112 (mapa =..., posicao = 5,10,
quantidade = 2)
fogefoge.rb:remove:112 (mapa =..., posicao = 5,9, quantidade = 3)
fogefoge.rb:remove:112 (mapa =..., posicao = 5,8, quantidade = 4)
```

Nesse instante, na nova linha `109`, ele encontra que quantidade vale `0` e para a recursão. Ao retornar de cada uma das invocações (da recursão), o jogo continua normalmente. Podemos testá-lo e ver que realmente funciona:

```
...
X X XXX   X XXX X
X  HXXX X X   X X
XXX*FXXFX   X   X
...
X X XXX   X XXX X
X   XXX X X   X X
XXXH    X   X   X
```

Recursão é o ato de uma função chamar a si mesma. Base da recursão é o critério onde essa recursão para, caso contrário ela executaria indeterminadamente o que poderia causar um estouro da pilha de execução.

Repare como nosso código final não possui mais o laço, utiliza a recursão e a pilha de execução para que o acumulador (`quantidade`) conte quantas casas foram limpas no mapa:

```
def remove mapa, posicao, quantidade
    if quantidade == 0
        return
    end
    posicao = posicao.direita
    mapa[posicao.linha][posicao.coluna] = " "
    remove mapa, posicao, quantidade - 1
end
```

Repare que fomos capazes de executar a recursão e a iteração (o laço) para atingir o mesmo resultado.

21.5 Base da recursão: distância quatro ou muro

Na prática, nossa bomba é forte demais, queremos que ela seja incapaz de explodir um muro. É muito incomum que um jogo permita esse tipo de explosão, portanto não podemos continuar se a posição atual for um muro.

Na implementação antiga, de um laço, poderíamos utilizar uma condição e um `break`:

```
def remove mapa, posicao, quantidade
    for direita in 1..quantidade
        posicao = posicao.direita
        if mapa[posicao.linha][posicao.coluna] == "X"
            break
        end
        mapa[posicao.linha][posicao.coluna] = " "
    end
end
```

Na nossa versão atual, recursiva, da função `remove`, basta adicionarmos uma nova base, se tem muro, para:

```
def remove mapa, posicao, quantidade
    if quantidade == 0
        return
    end
    posicao = posicao.direita
    if mapa[posicao.linha][posicao.coluna] == "X"
        return
    end
    mapa[posicao.linha][posicao.coluna] = " "
    remove mapa, posicao, quantidade - 1
end
```

Resultando em:

```
X X XXX   X XXX X
X  HXXX X X   X X
XXX*FXXFX   X   X
```

```
...
X X XXX   X XXX X
X   XXXFX X   X X
XXXH XX X   X   X
```

Podemos extrair agora uma função `execuca_remocao`:

```
def executa_remocao mapa, posicao, quantidade
    if mapa[posicao.linha][posicao.coluna] == "X"
        return
    end
    mapa[posicao.linha][posicao.coluna] = " "
    remove mapa, posicao, quantidade - 1
end

def remove mapa, posicao, quantidade
    if quantidade == 0
        return
    end
    executa_remocao mapa, posicao.direita, quantidade
end
```

21.6 Recursão para todos os lados: busca em profundidade

Por mais que nossa bomba respeite muros, nosso efeito ainda é bem estranho. Tradicionalmente, em jogos, uma bomba explode para todos os lados ao mesmo tempo, andando, por exemplo, 4 casas no máximo para qualquer direção, que é seu alcance.

Dado o mapa a seguir, o `mapa4.txt`:

```
XXXXXX
XHXXXX
X*FX X
XXFX X
XFFFFX
XXXXXX
```

A explosão *anda* quatro casas, de todas as maneiras possíveis. Note quão longe ela deveria conseguir chegar, somente até os fantasmas marcado com o número 4 no mapa a seguir:

```
XXXXXX
XHXXXX
X*1X X
XX2X X
X4345X
XXXXXX
```

Em outras palavras somente um fantasma sobreviveria. Como calcular isso? Queremos andar quatro de distância, por *todos os caminhos possíveis*, a partir de nossa bomba.

Uma solução inicial, com que temos que tomar um cuidado, é que não podemos fazer dois `for`, de `-4` a `+4`:

```
for movimento_linha in -4..4
    for movimento_coluna in -4..4
        linha = nova_posicao.linha + movimento_linha
        coluna = nova_posicao.coluna + movimento_coluna
        mapa[linha][coluna] = " "
    end
end
```

O movimento de nosso herói resultaria em:

```
XXXXXX
X XXXX
XH X X
XX X X
X    X
XXXXXX
```

Implementação completamente errada. Explodimos para todos os lados. Não é essa a descrição da explosão. Devemos percorrer todos os corredores em quatro de distância do ponto de origem, e limpar todos eles, nada mais.

Vamos olhar para nossa implementação recursiva:

```
def remove mapa, posicao, quantidade
    if quantidade == 0
        return
    end
    executa_remocao mapa, posicao.direita, quantidade
end
```

Em vez de tentarmos explodir somente a direita, tentamos as quatro direções, isso é, tentamos andar quatro de distância em todos os caminhos possíveis: estamos buscando todas as casas que conseguimos alcançar com quatro passos.

```
def remove mapa, posicao, quantidade
    if quantidade == 0
        return
    end
    executa_remocao mapa, posicao.direita, quantidade
    executa_remocao mapa, posicao.esquerda, quantidade
    executa_remocao mapa, posicao.cima, quantidade
    executa_remocao mapa, posicao.baixo, quantidade
end
```

Claro, adicionamos as funções de direção ao herói:

```
class Heroi
    # ...
    def cima
        calcula_nova_posicao "W"
    end
    def esquerda
        calcula_nova_posicao "A"
    end
    def baixo
        calcula_nova_posicao "S"
    end
end
```

Carregamos o quarto mapa:

```
def joga(nome)
    mapa = le_mapa(4)
    # ...
end
```

E agora tentamos rodar:

```
XXXXXX
XHXXXX
X*FX X
XXFX X
XFFFFX
XXXXXX
Para onde deseja ir?
S
XXXXXX
X XXXX
XH X X
XX XFX
X    X
XXXXXX
```

21.7 Resumindo

Vimos neste capítulo como implementar o algoritmo de destruição de um fantasma, para isso refatoramos mais nosso código, implementamos um laço através de um loop e de invocar a própria função, a recursão.

Encontramos a base da recursão quando percebemos que a pilha de execução estourava à medida que entrava em um laço infinito (ela se invocava até estourar). E com a recursão foi muito simples permitir andar quatro passos para todos os lados: foi possível buscar todos os quadrados do mapa que estão a até quatro passos distância, uma busca chamada *busca em profundidade*.

Capítulo 22

Exercícios extras: jogo do foge-foge

Aqui você encontra exercícios extras para soltar sua imaginação e praticar lógica de programação na linguagem Ruby.

22.1 Melhorando o Jogo do foge-foge

1) Suporte tanto letras maiúsculas quanto minúsculas ao se movimentar no mapa.

2) Se o jogador não pode se movimentar, perdeu o jogo. Para verificar se ele não pode se movimentar, crie uma função que confira se pelo menos uma das quatro posições é válida. Se pelo menos uma for válida, pode se movimentar, caso contrário, não pode.

3) Aumente os pontos do jogador: 5 pontos para cada movimento efetuado.

4) Crie um novo fantasma: ele nunca tenta se afastar do herói. Para isso, ele vê: se o herói está acima, não vai pra baixo. Se está abaixo, não vai para cima etc. Note que por vezes o fantasma poderá decidir ficar parado mesmo que haja uma casa ao lado dele.

5) Grave e leia de um arquivo quem é o jogador que obteve mais pontos.

6) Caso o arquivo de rank não exista, considere que não há ninguém no rank.

7) Pergunte qual a dificuldade que o jogador deseja, fácil, normal ou impossível. No modo fácil o jogador anda dois passos e os fantasmas um. No impossível, os fantasmas andam dois e o herói um. No modo fácil o jogador ganha somente 1 ponto por movimento, no normal 5 pontos e no difícil 20 pontos. Para implementar essa funcionalidade, você precisa armazenar tanto a dificuldade quanto o número do turno. Se o turno for par ou ímpar você sabe quem deve se mover.

8) Se o número de fantasmas for zero, o jogador ganha e leva `10000` pontos.

9) O mapa a seguir apresenta um bug. Se o fantasma não tem para onde ir, ele desaparece. Queremos que, caso ele não tenha para onde ir, ele deve ficar parado:

```
XXXXXX
XHXXXX
X XXFX
X XXFX
X XXFX
XXXXXX
```

22.2 Outros desafios

1) Grave em um arquivo os 10 jogadores que tiveram mais pontos.

Para manter o padrão de "outros desafios", podia ter pelo menos 2 nesta lista....]

Capítulo 23

Instalando o Ruby

A instalação em cada plataforma é feita de maneira diferente. No final, para testar a versão do Ruby disponível você pode executar o comando:

```
ruby -v
```

23.1 Instalação no Windows

Você pode utilizar o projeto Rubyinstaller para instalar o Ruby em um ambiente Windows. http://rubyinstaller.org/

Após instalado, no menu Iniciar você encontra o *Start command prompt with Ruby*, onde será inicializado um terminal do Windows (*command prompt*) com o comando *ruby* disponível. Vá para o seu diretório de trabalho usando o comando *cd* e pronto, *ruby* neles!

23.2 Instalação no Linux

Para Debian e Ubuntu a instalação é direta:

```
sudo apt-get install ruby-full
```

Para outras versões de Linux siga o tutorial específico na documentação do Ruby em:

https://www.ruby-lang.org/en/documentation/installation/

23.3 Instalação no Mac OS X

A partir da versão Maverick do Mac OS X, o Ruby 2.0 já vem instalado. Basta abrir a aplicação chamada `Terminal`. Já as versões Mountain Lion, Lion e Snow Leopard trazem o Ruby `1.8.7`. Tudo o que fizemos neste livro é independente da versão do Ruby e compatível desde a `1.8.7`, mas se desejar instalar uma versão mais recente você pode utilizar o gerenciador de pacotes `HomeBrew` e executar:

```
brew install ruby
```

23.4 O editor de texto

Ao escrever este material utilizamos o editor de texto Sublime. Ele funciona para as plataformas Mac OS X, Linux e Windows e é a ferramenta sugerida ao acompanhar o livro.

Como alternativa, no ambiente Mac OSX, outra opção bastante popular no começo é o Textmate.

Já no ambiente Linux, uma alternativa simples é o Gedit.

No Windows, jamais utilize o bloco de notas (*notepad*). Se não por diversos outros problemas, ele traz a dificuldade de salvar o arquivo com a extensão adequada. Como alternativa ao Sublime tente soluções como o Notepad++, jEdit etc.

Uma solução que funciona nas três plataformas, mas mais pesada e que seria um segundo passo, é o Aptana, a IDE baseada no Eclipse.

Não recomendo o uso de Emacs ou Vi como editores de texto durante o aprendizado deste material em ambientes Mac OSX ou Linux nesse instante. O foco deste livro é o aprendizado de linguagem de programação, que já trará desafios suficientes para essa etapa. No futuro, sinta-se à vontade para encontrar qual a ferramenta que se encaixa melhor com sua maneira e a de sua equipe de trabalhar.

Capítulo 24

Como continuar

24.1 Praticar

Os conceitos de programação são fundamentais para o dia a dia de um desenvolvedor. Claro que o domínio vem com o tempo. Simular um código de cabeça, bater o olho e entender onde estão os possíveis erros, tudo isso vem com anos de experiência. Assim como a comida de nossos avôs é deliciosa pois eles prepararam o mesmo prato toda semana por 60 anos, não adianta tentar pular passos: pratique. Escreva programas, joguinhos, simuladores etc.

O caminho mais comum é o de praticar bastante, sentir a necessidade de aprender mais estruturas, depois algoritmos, reforçar a maneira de trabalhar com Orientação a Objetos e então ter a visão de como outras linguagens de programação funcionam.

24.2 Estrutura de dados e algoritmos

Para trabalhar com dados na memória, é importante entender um pouco como funcionam estruturas de dados básicas. Neste material vimos como funcionam um array, um dicionário associativo, e até mesmo uma pilha (a pilha de execução). Com o passar do tempo, conhecer melhor essas e outras estruturas ajudará a entender melhor o que acontece na memória e como resolver problemas lógicos de maneira elegante, sem tentar reinventar a roda.

Ao mesmo tempo, para utilizar tais estruturas de dados de maneira ótima, aprendemos determinados algoritmos que resolvem problemas de diversas maneiras possíveis, sempre tirando proveito do que cada uma dessas estruturas traz de melhor.

24.3 Orientação a Objetos?

O foco deste material não reside no ensino de estruturas de dados, mas decidimos ir um pouco além e mostrar métodos em classes, uma vez que tudo (sim, tudo) em Ruby é um objeto, além de algumas boas práticas e code smells ligados ao assunto. Existe muito mais no mundo de orientação a objetos: a facilidade de manutenção de nosso código depende muito de conhecer o seu bom uso. Um ponto interessante de continuação para seus estudos em um tópico mais avançado do que a introdução a programação atual é o mundo de orientação a objetos mas não se esqueça, todo ensinamento deve conter o lado bom e ruim de diversas funcionalidades demonstradas. No caso de Ruby, isso engloba o aspecto orientado a objetos, inicializadores, classe aberta, métodos de classe, herança, mixins, polimorfismo (como através de duck typing) e muito mais.

24.4 Competições de programação da ACM

Táticas como a apresentada aqui, de utilização de matrizes para resolver um problema de lógica, são extremamente comuns nas competições de programação no mundo afora.

Todo ano é realizada a ACM ICPC, a competição mais famosa da área, cujos exercícios se focam no desenvolvimento de algoritmos para resolver di-

versos tipos de problemas. A prova classificatória para o mundial é realizada no Brasil com o nome de Maratona de Programação. É necessário ser aluno de alguma faculdade brasileira (de qualquer área) para poder participar.

O estudo para essas provas serve como grande desafio para nossas habilidades lógicas, além de aprendermos mais a fundo sobre a linguagem que utilizamos para resolver seus problemas. Por exemplo, os casos do `+1` e `-1` que utilizamos são um perigo para uma aplicação: esquecer um deles pode quebrar o jogo e com isso perder um usuário completamente. O que fazer? Uma tática simples, típica de maratona, é se proteger das bordas. Se ao ler um mapa colocarmos uma barreira de `X` em cada um de seus lados, não precisamos mais validar com os `ifs` das bordas, afinal elas *sempre* têm um `X`! Técnicas como essa são úteis para quem programa um código mais matemático.

Por outro lado, os desafios lógicos incentivam nossa capacidade de resolver problemas e de descrevê-los em comandos para o computador. Se tiver a chance, treine e participe! Nem que seja pelo prazer de aprender. Não se assuste com o nível de dificuldade dos exercícios, comece com os mais leves e à medida que estuda estrutura de dados, algoritmos e outros tópicos, os exercícios mais avançados vão se tornando mais naturais.

24.5 Outra linguagem?

Após se sentir familiarizado o suficiente com uma linguagem, uma opção de muitos programadores é dar uma olhada, ou até mesmo se aventurar de corpo e alma, em outra linguagem de programação. Se Ruby foi sua primeira linguagem, você pode aprender agora Javascript para aplicar programação aos navegadores web, Java ou C# para desenvolver aplicativos mobile ou web, ou a linguagem C que trará um conhecimento mais avançado de como o computador e as linguagens funcionam. PHP permite a criação de sites, e Python é bastante utilizado na criação de programas a serem rodados no terminal, serviços etc. Além disso, o Java é base para o desenvolvimento Android, enquanto Swift é para o desenvolvimento iOS.

O mundo das linguagens é muito grande, dê um passo por vez. Somente quando se sentir bem o suficiente com a questão de programação escolha uma

próxima linguagem para saciar sua curiosidade. Avance em sua carreira de acordo com o caminho que pretende seguir.

24.6 Compartilhe e boa jornada

Compartilhe seu código com outros. Use ele para pegar feedback de outros programadores e aprenda com o código deles. Sites como o GitHub e softwares de controle de versão (como o git) ajudam com essa tarefa. Ao criar sua conta gratuita no GitHub e compartilhar seu código como open source, poderá facilmente conversar com outros amigos programadores sobre ele e, quem sabe, até contribuir com outros ou receber contribuições em seus projetos.

Bons estudos, boa jornada e boa carreira! Fico no aguardo pelo Twitter @guilhermecaelum de saber como a programação mudou sua vida.

Índice Remissivo